Temporale Prüfung von Software-Funktionen

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von:	Andreas Schulze
aus (Geburtsort):	Stendal

eingereicht am:	25. Oktober 2010
mündliche Prüfung am:	25. August 2011
Referenten:	Prof. Dr.-Ing. Wilfried Daehn Prof. Dr.-Ing. Walter Schumacher Prof. Dr.-Ing. Hans-Michael Hanisch

2011

Veröffentlichungen über den Inhalt der Arbeit sind nur mit schriftlicher Genehmigung der Volkswagen AG zugelassen.

Die Meinungen, Ergebnisse und Schlüsse in dieser Dissertation sind nicht notwendigerweise die der Volkswagen AG.

Kurzfassung

In der Automobilelektronik kommen im Bereich der Funktions- und Software-Entwicklung zunehmend modellbasierte Entwicklungswerkzeuge zum Einsatz. Diese Entwicklungswerkzeuge (z. B. ASCET-SD und MATLAB/Simulink) ermöglichen es, schnell die Anforderungen an eine Fahrzeugfunktion (z. B. Geschwindigkeitsregelung) zu modellieren, zu simulieren und aus dem grafischen Funktionsmodell heraus den Programmcode zu generieren. Die modellbasierte Software-Entwicklung im Zusammenspiel mit effizienten Entwicklungsmethoden und Werkzeugen hat in den letzten Jahren zu einer Verschiebung der Aufgabenverteilung zwischen Fahrzeughersteller und Zulieferer geführt. Software für eingebettete Systeme wird aufgrund der damit verbundenen Innovationen und dem daraus resultierenden Differenzierungsmerkmal zu anderen Fahrzeugherstellern zunehmend durch die Fahrzeughersteller selbst entwickelt. Die Gründe hierfür sind die leicht lesbaren grafischen Funktionsbeschreibungen zur Spezifikation der Software und die schnelle Erlebbarkeit der Funktion auf Prototypensystemen. Entscheidend ist jedoch der aus der Codegenerierung entstehende Vorteil in der leichten Portierbarkeit der Software und der dadurch ermöglichten Übertragung von Fahrzeugfunktionen auf verschiedene Zielsysteme. Letztendlich führt dies zu einer Reduzierung der Entwicklungskosten.

Im Rahmen der modularen Funktionsintegration für Motorsteuergeräte-Software wird nach Wegen gesucht, um neue Produktideen bei der verteilten Software-Entwicklung zwischen Fahrzeughersteller und Motorsteuergerätezulieferer sicher in Serie zu bringen. Bei der Entwicklung von Software-Funktionen für Motorsteuerungen durch den Fahrzeughersteller müssen diese in die Steuergeräte-Software des Systemlieferanten integriert werden. Dabei muss der Systemlieferant die Echtzeitfähigkeit des Gesamtsystems sicherstellen. Dies schließt die selbstentwickelten Systemkomponenten und die fremdentwickelten Software-Komponenten des Fahrzeugherstellers ein. Um die geforderte Qualität des Gesamtsystems sicherzustellen, müssen Beschreibungen der Software-Komponenten mit den Beschreibungen der Systemkomponenten zusammengefasst werden. Die Beschreibung des Zeitverhaltens von fremdentwickelten Software-Komponenten erfolgt auf Basis von Testfällen, die die rechenintensiven Programmpfade stimulieren. Der Systemhersteller soll damit die durchschnittliche und maximale Laufzeit der Funktion durch Messung verifizieren können. Die Laufzeit der Software hängt dabei von vielen Faktoren ab, wie dem Zustand des Systems und der Funktion der Software. Zum Beispiel braucht eine Geschwindigkeitsregelung für die Funktion SET[1] eine andere Laufzeit als für die Funktion RESUME[2]. Dies resultiert unter anderem aus den unterschiedlichen Befehlen, die auf verschiedenen Programmpfaden abgearbeitet werden müssen.

Der Nachweis der Echtzeitfähigkeit setzt eine vollständige Verifikation der Software und der Hardware voraus. Damit ist auch eine Berücksichtigung aller Systemzustände für alle Eingaben notwendig. Die praktische Systemverifikation wird derzeit so vollzogen, dass das System durch Tests in den verschiedenen Entwicklungsphasen analysiert wird. Aufgrund der Software-Komplexität und des damit verbundenen hohen Aufwands für einen vollständigen Test eines Programms werden meist Kompromisse eingegangen, so dass zum Beispiel nicht alle

[1] Speichern und Halten der vom Fahrer gewählten Geschwindigkeit
[2] Wiederaufnahme der zuletzt gespeicherten Geschwindigkeit

Programmpfade durchlaufen werden. Stattdessen werden repräsentative Testfälle erstellt und das Programm danach beurteilt. Dies bedeutet, dass nicht alle echtzeitbeeinflussenden Faktoren getestet werden. Insbesondere ist nicht sichergestellt, dass durch Tests die obere Laufzeitgrenze, das heißt, die maximale Zeit zur Erfüllung der Aufgabe, ermittelt wird und hierfür ein Testfall angegeben werden kann. Die Forderung, dass alle Funktionen die vorgegebenen Zeiten garantiert einhalten können, ist somit nicht mit Sicherheit erfüllt.

Im Fokus dieser Arbeit steht die Einführung der Entwicklung von Software-Funktionen durch den Fahrzeughersteller und die damit verbundenen Anforderungen an eine teilweise fremdentwickelte Software für Steuergeräte. Das konkrete Ziel dieser Arbeit ist, die analytische Qualitätssicherung so zu erweitern, dass durch dynamische, strukturorientierte Tests vollständige und verlässliche Ergebnisse zur Bewertung des zeitlichen Verhaltens von Fahrzeugfunktionen möglich sind, ohne den Aufwand etablierter Test wesentlich zu erhöhen. Dafür werden die Elemente der analytischen Qualitätssicherung untersucht sowie der organisatorische und technische Rahmen, in dem sich die Laufzeitanalyse einbetten muss, betrachtet. Der Stand der Technik zum Software-Test wird vorgestellt und die bestehenden Unzulänglichkeiten zusammengefasst.

Der eigene Beitrag liegt in der Entwicklung des temporalen Prüfkonzeptes, um die bestehenden Unzulänglichkeiten bei der analytischen Qualitätssicherung des zeitlichen Verhaltens abzustellen. Das Prüfkonzept stellt eine Vorgehensweise zur Lösung dieser Problemstellung bereit, die Ausarbeitung erfolgt dann in der Implementierung als Werkzeug und Verfahren. Ziel ist es, das zeitliche Verhalten der Software frühzeitig abzusichern. Hieraus wird eine Strategie zum temporalen, dynamischen, strukturorientierten Test formuliert und die notwendigen Anforderungen an den Software-Modultest abgeleitet. Es wird als neuer Ansatz ein temporaler Regressionstest eingeführt, um die Aufwendungen für das Testen zu reduzieren. Dazu wird ein Messprozess zur Erfassung der Laufzeit definiert und ein eigenständiges Konzept zur Analyse von Laufzeitmessdaten vorgestellt. Die Implementierung dieses Prüfkonzeptes in Form von Verfahren und Werkzeugen wird für das Referenzbeispiel eines Motorsteuergerätes zur Verfügung gestellt, um eine Evaluierung des Prüfkonzeptes zu ermöglichen. Hierfür werden übertragbare Ansätze zur Datengewinnung und zur Datenanalyse entworfen, die an ausgewählten Fallstudien diskutiert werden. Das Ziel der hier dargestellten Fallstudien ist es, die Wirksamkeit des temporalen Prüfkonzeptes und dessen Implementierung in Techniken und Werkzeuge zu demonstrieren.

Um eine Einordnung der Arbeit zu ermöglichen, werden vorher die notwendigen Grundlagen sowie die Probleme der Laufzeitbestimmung für eine Software-Funktion diskutiert. Zum Vergleich und zur Abgrenzung zu anderen Verfahren wird ein Überblick über den Stand der Technik zur Laufzeitanalyse aus aktuellen Forschungsprojekten sowie aus der kommerziellen Entwicklung zusammengestellt. Die Bewertung, die vorgenommen wird, basiert auf Faktoren, die vor allem die industrielle Verwendung und die daraus resultierenden Anforderungen an Methoden und Verfahren zur Laufzeitanalyse von Motorsteuergeräte-Software einbeziehen.

Inhaltsverzeichnis

1 Einführung

Software im Fahrzeug beschäftigt die Automobilindustrie (Fahrzeughersteller und Zulieferer) schon seit einigen Jahren, zuerst als erforderlicher Funktionsbestandteil mit der Einführung von Steuergeräten, um immer komplexere Regel-, Diagnose- oder Überwachungsfunktionen zu realisieren, anschließend als eine treibende Kraft in der Automobilindustrie, die zu einem wahren Innovationsschub geführt hat. Software und Elektronik sind aus keinem Fahrzeug mehr wegzudenken, das heutigen Komfort- und Sicherheitsansprüchen sowie Verbrauchs- und Emissionsanforderungen genügen muss. Auch zukünftig ist damit zu rechnen, dass Umfang und Komplexität von Software im Fahrzeug steigen, vor allem getrieben durch neue gesetzliche Bestimmungen sowie Leistungs- und Qualitätsanforderungen. Völlig neue Funktionen des Fahrzeugs können und müssen so durch Software bereitgestellt werden.

Diese Fahrzeugfunktionen werden durch zusätzliche elektronische Komponenten, durch neue Software-Funktionen in den Steuergeräten und durch Vernetzung von Steuergeräten ermöglicht. Unter Software-Funktion werden Funktionsmerkmale des Fahrzeugs verstanden, die durch Software realisiert werden. Diese Funktionen werden vom Benutzer, etwa dem Fahrer, direkt oder indirekt als Merkmale des Fahrzeugs wahrgenommen. Die Innovationskraft und die Qualität dieser Merkmale machen im Endeffekt die Wettbewerbsdifferenzierung aus.

Neben dem Potenzial, das Software im Zusammenspiel mit Elektronik mit sich bringt, bestehen auch Risiken. Die Risiken sind vor allem erst mit dem Anstieg der Software-Umfänge und den Folgen von fehlerhaften Software-Funktionen bewusst geworden. Sie machten auf Lücken in den Prozessen, Methoden und Verfahren der Fahrzeugentwicklung aufmerksam. Die bestehenden Prozesse, Methoden und Verfahren müssen den Belangen einer Software-Entwicklung gerecht werden.

Dies bringt mit sich, die zusätzlichen Aufwendungen für Software-Entwicklung, Integration und Test einzuplanen, in kurzen Entwicklungszeiten den hohen Qualitätsanforderungen gerecht zu werden und die späte Systemintegration einzukalkulieren, um Fehler frühzeitig zu entdecken.

Dies verlangt nach abgestimmten systematischen Entwicklungs-, Integrations- und Testphasen zwischen den Entwicklungspartnern. Ansätze zur Beherrschbarkeit müssen interdisziplinäre, unternehmensübergreifende Entwicklungs- und Prüfprozesse bereitstellen, die die Software-Entwicklung als eigenständiges Thema betrachten und durch geeignete Werkzeuge unterstützen, um die Qualitätsziele zu erreichen.

Dazu gehören:

- Das Definieren von Standards/Normen [81] für sicherheitskritische Systeme. Diese Sicherheitsnormen beschreiben Maßnahmen, um die notwendige Sicherheit zu erreichen. Hierbei wird unterschieden in Entwicklungsmethoden, Werkzeuge, Implementierung[1], Verifikation[2] und Validation[3] des Systems.
- Neben diesen Maßnahmen werden Prozesse bzw. Vorgehensmodelle zur Beschreibung des Entwicklungsablaufs eingesetzt, um Abläufe zu strukturieren (s. Abschn. 1.1).

[1] Umsetzung einer formalisierten Beschreibung (Spezifikation)
[2] nachprüfen, die Richtigkeit einer Umsetzung beweisen
[3] bekräftigen, die Gültigkeit einer Umsetzung zu einer Spezifikation erklären

- Die Prozesse müssen durch geeignete Werkzeuge für die Software-Entwicklung und Prüfung unterstützt werden. Bevorzugt werden Modellierungs- und Simulationswerkzeuge für die Entwicklung eingesetzt sowie Hardware-in-the-Loop-Systeme, Prüfstände und Versuchsfahrzeuge für die Prüfung (s. Abschn. 1.1.2).

Nachfolgend werden die wesentlichen Zusammenhänge zu den Hintergründen dieser Arbeit dargelegt. Es werden Unzulänglichkeiten und Ansätze in der Qualitätssicherung bei teilweise fremdentwickelten Software-Funktionen beschrieben. Der Themenschwerpunkt, die Absicherung des temporalen Verhaltens von Software-Funktion im Fahrzeug, wird dargestellt, hieraus wird die Zielstellung formuliert.

1.1 Klassischer Entwicklungsablauf

Der klassische Entwicklungsablauf für die Entwicklung eingebetteter Software zwischen Fahrzeughersteller und Zulieferer ist in Abb. 1.1 dargestellt. Das Vorgehensmodell ist ein strukturierter, in Phasen aufgeteilter Prozess, in dem sich konstruktive und analytische Schritte abwechseln. Ein Prozessschritt ist eine abgeschlossene Folge von Tätigkeiten, die als Ergebnis ein Artefakt liefern, das in einem anderen Prozessschritt weiterverwendet wird. Häufig wird zur Beschreibung des Entwicklungsprozesses das V-Modell [7] benutzt. Auf der linken Seite des V-Modells werden die konstruktiven Schritte abgebildet. Das 1. Artefakt, das entsteht, ist die Anforderungssammlung.

1. **Analyse der Anforderungen:** Dieser Schritt dient zur Erstellung der Spezifikation für das gesamte elektronische Fahrzeugsystem, das 2. Artefakt ist die Systemspezifikation.
2. **Systementwurf:** Dieser Schritt dient zur Erstellung der Spezifikation der einzelnen Hardware-Komponenten (Steuergerät, Sensoren, Aktoren, Busse) und der einzelnen Software-Funktionen, das 3. Artefakt ist die Hardware-Spezifikation und Software-Spezifikation.

Im klassischen Entwicklungsprozess werden diese Aufgaben durch den Fahrzeughersteller ausgeführt. Die Umsetzung der Hardware- und Software-Spezifikation erfolgt durch die Zulieferer. Erfolgt die Systementwicklung und Herstellung durch einen Zulieferer, wird dieser als Systemlieferant bezeichnet.

3. **Software-Entwurf:** Hier wird sich mit der Aufteilung der Software-Funktion auf einzelne Module beschäftigt, das 4. Artefakt ist die Modulspezifikation. Im Folgenden soll lediglich die Software-Entwicklung betrachtet werden.
4. **Implementierung:** Dieser Schritt ist die Umsetzung der Spezifikation in Programmcode. Das Ergebnis ist das 5. Artefakt, der Programmcode.

Die rechte Seite des V-Modells wird durch Integrationsschritte der einzelnen Module zu einem Programmstand und die Integration in die Hardware sowie die Vernetzung der einzelnen Hardware-Komponenten gebildet. Hierauf werden die analytischen Schritte mit den jeweiligen Testphasen abgebildet. Die Steuergeräte-Software (das Programm) enthält zunächst nur eine Grundbedatung. Das können stationäre Prüfstandsdaten sein. Diese müssen bzgl. Fahrbarkeit (über Applikationsparameter / Freiheitsgrade zur Anpassung des Funktionsverhaltens) noch angepasst werden. Der Vorgang wird allgemein als Applikation bzw. Kalibrierung bezeichnet.

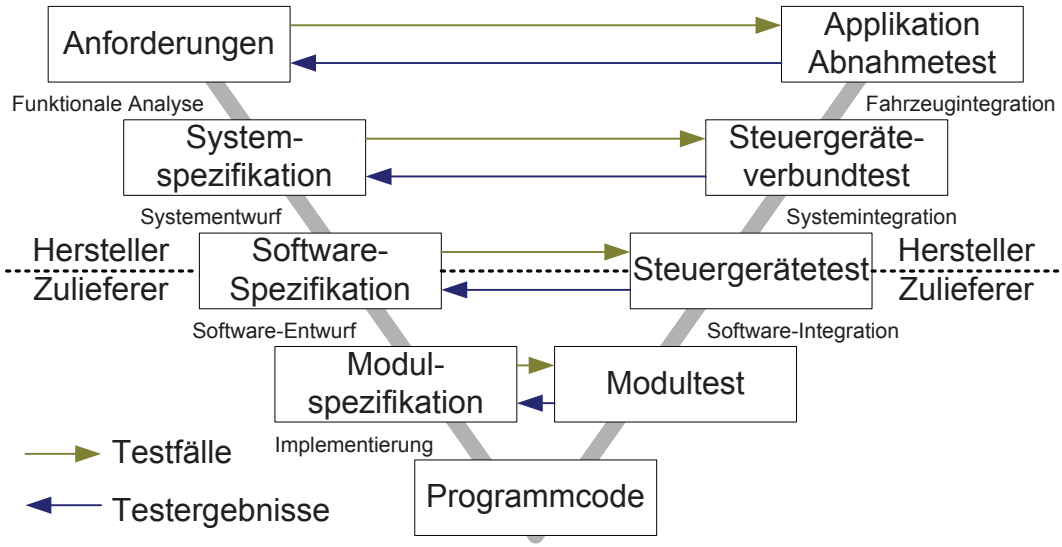


Abb. 1.1: Entwicklungsablauf zwischen Fahrzeughersteller und Zulieferer

Die Applikation der Steuergeräte-Software im Fahrzeug nimmt eine hybride Rolle zwischen den konstruktiven und analytischen Schritten ein. Zum einen werden die Software-Funktionen mit fahrzeugspezifischen Daten belegt, zum anderen werden diese Daten im Zusammenspiel mit der Software geprüft. Die Testphasen bilden einen eigenständigen Bestandteil im Entwicklungsprozess zur Qualitätssicherung.

5. **Modultest:** Dieser Schritt dient zur Verifikation von Modulen einer Software-Funktion entsprechend der Modulspezifikation. Das 6. Artefakt ist das Modul als verifizierter Programmcode oder der Objektcode.
6. **Steuergerätetest bzw. Integrationstest:** Hier werden verschiedene Tests zusammengefasst, die dazu dienen, die voneinander abhängigen Hardware-Komponenten und Software-Funktionen eines Systems im Zusammenspiel miteinander zu testen. Das 7. Artefakt ist der flashbare Programmstand mit der Grundbedatung für ein Steuergerät.
7. **Steuergeräte-Verbundtest bzw. Systemtest:** Dieser Schritt dient dazu, die voneinander abhängigen Komponenten des gesamten Systems im Zusammenspiel miteinander zu testen.
8. **Applikation und Abnahmetest bzw. Akzeptanztest:** Der abschließende Schritt umfasst die Applikation und den Abnahmetest, hier wird das System gegen die Anforderungen getestet, der Akzeptanztest ist die Validation des Systems.

1.1.1 Einführung der modellbasierten Funktionsentwicklung

Mit der Einführung der modellbasierten Funktionsentwicklung wurden neue Entwicklungsmethoden bereitgestellt, die die Wirtschaftlichkeit und Qualität der Software-Entwicklung und damit des Produktes steigern sollen.

- Die Software-Spezifikation wurde um die Anteile des Funktions- bzw. Software-Entwurfs erweitert. Dies erfolgt durch Einsatz von Modellierungs- und Simulationswerkzeugen, mit denen mathematische Modelle und Beschreibungsformen wie Blockdiagramme, die ausführbare (simulierbare) Spezifikationen darstellen (sogenannte Funktionsmodelle), erstellt werden konnten. Mithilfe des Funktionsmodells sollen Software-Fehler durch unvollständige und mehrdeutige Spezifikationen vermieden werden.

- Zur Prüfung des Modells wurden auf der linken Seite des Vorgehensmodells zwei Testphasen eingeführt, zunächst mit dem vorrangigen Ziel die Spezifikation der Software-Funktionen zu prüfen. Die Prüfung erfolgt in der Simulationsphase durch die Ausführung des Funktionsmodells auf dem PC. Dieser sogenannte Offline-Test stimuliert die Eingänge des Modells mit Testdaten und lässt eine Überprüfung der Ergebnisse des mathematischen Modells zu. Hierdurch können Spezifikationsfehler erkannt werden. Die Prüfung der Echtzeitfähigkeit erfolgt in der Prototypenphase. Eine neue Software-Funktion kann hier im realen Fahrzeug oder am Prüfstand getestet werden. Zur Prüfung unter Echtzeitbedingungen wird die Software auf Prototypenrechnern ausgeführt, die Testdaten kommen von den Sensoren des Fahrzeugs, die Ergebnisse des Modells beeinflussen die Aktoren des Fahrzeugs direkt oder indirekt. So lassen sich Entwurfsfehler frühzeitig aufdecken.

Das 3. Artefakt, die Software-Spezifikation, wird zu einer ausführbaren Spezifikation bzw. ein ausführbares Lastenheft (Funktionsmodell), das einen Teil der Implementierung bereits umsetzt.

- Die Übertragung der Spezifikation (Modell) in die Implementierung wurde in Werkzeuge integriert. Das Modell liegt meist als Fließkomma-Beschreibung vor. Diese erfordert eine Übertragung in eine Festkomma-Beschreibung zur Speicheroptimierung und Zielhardwareanpassung. Anschließend erfolgt die automatische Codegenerierung aus dem implementierten Modell, d. h. die Erstellung des Programmcodes.

Die modellbasierte Funktionsentwicklung verschiebt die Schnittstelle zwischen Hersteller und Zulieferer gegenüber dem klassischen Steuergeräte-Entwicklungsprozess. Implementierungsaufgaben werden zunehmend durch den Fahrzeughersteller in Form des Funktionsmodells übernommen. Der Fahrzeughersteller verfolgt damit ein weiteres Ziel, die Entwicklung von Software-Komponenten. Ökonomisch sinnvoll wird Software-Entwicklung erst bei der Wiederverwendung von Software durch Reduzierung von Entwicklungs- und Testaufwand. Dies ist mit der Erstellung von Software-Komponenten [35] möglich. Software-Komponenten sind einzelne Module, die unabhängig von der Hardware und dem Betriebsystem eingesetzt werden können. Sie verfügen über definierte Schnittstellen zur Kopplung mit anderen Komponenten. Die Basis hierfür bildet eine komponentenbasierte Software-Architektur[4] [88].

Bei der Entwicklung von Software-Funktionen entstehen unterschiedliche Entwicklungsartefakte mit unterschiedlichem Produktreifegrad. Je nach Art der Software sind vom Fahrzeughersteller die Punkte der Tabelle 1.1 gegeneinander abzuwägen, um zu entscheiden, nach welcher Entwicklungsphase das jeweilige Artefakt an den Systemlieferanten zur Implementierung in einen Programmstand übergeben werden soll. Aus diesem Vorgehen resultieren mehrere Konsequenzen, die weitest reichende betrifft die Qualitätssicherung, die im nachfolgenden Abschnitt erläutert werden soll.

Tabelle 1.1: Reifegradparameter zur Festlegung des Austauschartefakts

Vorteile	Nachteile
- wirtschaftliche Vorteile	- Risikoübernahme
- Know-How-Schutz	- höherer Testaufwand mit steigenden Qualitätsanforderungen[a]
- Wettbewerbsvorteile	
- kürzere Entwicklungszyklen	- Bindung von eigenem Personal für Software-Entwicklung und Qualitätssicherung
- Steuerung von Kompetenzen	
- Aufbau alternativer Software-Lieferanten	

a - abhängig vom Produktreifegrad des zu implementierenden Artefakts

[4] Beschreibung der Komponenten und deren Beziehungen im Software-System

1.1.2 Übernahme der Qualitätssicherung

Die bisherige vollständige Eigenentwicklung des Systems durch den Systemlieferanten ermöglicht andere Ansätze und Verfahren als bei teilweiser Fremdentwicklung. Bei vollständiger Eigenentwicklung liegen die gesamten Strukturinformationen des Systems vor, das bedeutet, für die Software liegt der Quellcode offen. Die Qualitätssicherung kann durch einen:

- durchgängigen Entwicklungs- und Prüfprozess gestaltet werden,
- es liegen einheitliche Beschreibungen der Komponenten vor und
- die Analyse kann durch Automatisierung der Qualitätssicherungstechnik auf Basis der elektronisch verfügbaren Daten erfolgen und bestehende Risiken können leicht beurteilt werden.

Bei der teilweisen Fremdentwicklung der Software durch den Fahrzeughersteller müssen neue Ansätze zur Qualitätssicherung gefunden werden. Die Erbringung der geforderten Qualitätsnachweise muss durch die Software-Zulieferer (Fahrzeughersteller) erfolgen. Dies geschieht auf Basis eines Vertrages zwischen Fahrzeughersteller und Systemlieferant. Der Vertrag definiert die genauen Randbedingungen der Fremdentwicklung, um die geforderte Qualität des Gesamtsystems sicherzustellen. Wichtig ist hierbei, dass:

- die Verlässlichkeit der Ergebnisse sichergestellt ist und
- bei kritischen Anwendungen formal vollständige oder statistisch abgesicherte Ergebnisse vorliegen.

Dafür ist es notwendig, eine gemeinsame Komponenten-Beschreibung zu definieren, damit Aussagen bzw. Beschreibungen der fremdentwickelten Software-Komponenten mit den Ergebnissen der selbstentwickelten Systemkomponenten zusammengefasst werden können.

Für jede Software-Komponente, die durch den Systemintegrator (in den meisten Fällen ist das der Systemlieferant) als Objektcode in den Programmstand integriert werden soll, wird eine Verifikation der Software nach zwei Aspekten verlangt:

1. Verifikation funktionaler Eigenschaften: Dies ist die Prüfung der Korrektheit der Funktion. Diese wird durch einen klar definierten und gelebten Entwicklungsprozess sichergestellt, der dynamische Tests gegen die Spezifikation im Labor (Simulationsphase) und Fahrzeug (Prototypen- und Musterphase) enthält.
2. Verifikation nichtfunktionaler Eigenschaften: Für die Integration der Software-Komponente in ein Gesamtsystem müssen Qualitätseigenschaften wie Sicherheit, Zuverlässigkeit sowie Restriktionen wie Rechenzeit (Laufzeit)- und Speicherbeschränkungen der Funktion geprüft sein. Dies soll nicht nur die korrekte Funktion der Software, sondern auch deren Echtzeitfähigkeit sicherstellen.

Die Verifikation funktionaler und nichtfunktionaler Eigenschaften ist vertraglich so geregelt, dass eine Freigabe des gesamten Systems bei verteilter Entwicklung grundsätzlich erfolgen kann. Die gemeinsame Komponenten-Beschreibung enthält dafür, neben der Beschreibung der Komponenten-Schnittstellen, eine Beschreibung der maximal zulässigen Programmlaufzeit und dem maximal verfügbaren Speicher für die fremdentwickelte Software-Komponente.

1.2 Themenschwerpunkt

Im Vorfeld dieser Arbeit wurden von Volkswagen mehrere Studien [14, 15, 16, 17] in Auftrag gegeben. Die Durchführung dieser Studien wurde vom Institut für Datentechnik und Kommunikationsnetze (IDA) der Technischen Universität Braunschweig vorgenommen. Im Mittelpunkt dieser Studien stand der Software-Entwurfsprozess der Aggregateelektronik. Als

Ergebnis dieser Studien wurde weniger die funktionale Entwicklung (bzw. der Entwicklungsprozess) von Software-Funktionen kritisiert, als vielmehr die Entwicklung und Verifikation der nichtfunktionalen Eigenschaften. Es wurde herausgestellt, dass insbesondere die Sicherstellung der Echtzeitfähigkeit ein ungelöstes Problem der Software-Integration darstellt. Diese Studien waren damit der Grundstein der vorliegenden Dissertation.

Die Behandlung von nichtfunktionalen Eigenschaften bei der Entwicklung erfordert spezielle Techniken, die in den bestehenden Prozess integriert werden müssen. Ziel muss es sein, Fehler frühzeitig zu erkennen und zu beheben. Der Fehlerentdeckungs- und Fehlerbehebungsprozess sowie Strategien zur systematischen Fehlervermeidung für Software als Bestandteil des Fahrzeugs müssen Elemente des Entwicklungsprozesses sein.

Schwerpunkt Rechenzeitbeschränkung: Allgemein werden Systeme, die zeitlichen Anforderungen folgen, als Echtzeitsysteme bezeichnet (s. Abschn. 2.1). Die Anforderung an ein Echtzeitsystem beinhaltet neben der funktionalen Korrektheit auch die zeitliche Korrektheit (s. Abschn. 2.3). Die zeitlichen bzw. temporalen Zusammenhänge müssen der Spezifikation entsprechend umgesetzt und nachgewiesen werden, um die Echtzeitfähigkeit des Gesamtsystems sicherstellen zu können. Echtzeit-Software verlangt vom Hersteller die Berücksichtigung der zeitlichen Anforderungen in der Konstruktions- und Analysephase. Die Zeitanalyse zur Absicherung des temporalen Verhaltens einer Software ist ein Element der Software-Qualitätssicherung.

Die Herausforderung besteht darin, die obere Laufzeitgrenze eines Programms, d. h. die Worst-Case Execution Time (WCET), zu bestimmen. Dabei wird der Aufwand zur Laufzeitbestimmung durch die Komplexität des Programms und der Hardware bestimmt, auf der das Programm zur Ausführung gebracht wird. Der Aufwand zur vollständigen Analyse steigt exponentiell mit der Anzahl der Systemzustände und Eingangssignale, die den Programmablauf und damit die Programmlaufzeit beeinflussen. Die Gründe liegen in der datenabhängigen Ausführungszeit der Befehle eines Programms sowie im Einfluss der Hardwarearchitektur auf deren Ausführungszeit. Bei der Laufzeitbestimmung sind zwei Parameter zu beachten, zum einen muss die ermittelte obere Grenze sicher sein, um die Echtzeitfähigkeit garantieren zu können, und zum anderen darf diese obere Grenze nicht zu konservativ sein, um eine teure Überdimensionierung des Systems zu vermeiden.

Neben diesen Herausforderungen wird noch eine weitere Randbedingung aufgestellt, die durch die definierte gemeinsame Komponenten-Beschreibung bestimmt ist. Um ein Zusammenfassen der Aussagen bzw. Beschreibungen der fremdentwickelten Software-Komponenten des Fahrzeugherstellers mit den Ergebnissen der Systemkomponenten des Systemlieferanten zu ermöglichen, muss die Worst-Case Execution Time als Testfall, d. h. durch Vorgabe von Eingabedaten, beschrieben werden.

2 Grundlagen

Im Folgenden werden die wesentlichen Grundlagen bei der Entwicklung von Fahrzeugfunktionen eines Echtzeitsystems kurz zusammengefasst. Ein Anspruch auf eine umfassende Darstellung der Gebiete wird nicht erhoben. Es wird sich auf die Vorstellung der Grundlagen, Begriffe und Definitionen, soweit sie für das Verständnis der Arbeit notwendig sind, beschränkt. Die hierfür genutzte Begriffswelt ist angelehnt an vorhandene Standards und soweit es sinnvoll erscheint an deutschen Begriffen. Dieses Kapitel führt in die Problemstellung der Laufzeitbestimmung für eine Software-Komponente ein. Einleitend wird das Echtzeitsystem definiert. Die Realisierung eines Echtzeitsystems in Form eines eingebetteten Systems und eines Software-Systems wird am Beispiel einer Motorsteuerung dargestellt. Anhand dieser Realisierung soll die Laufzeitproblematik für das Gesamtsystem, das heißt die globale Laufzeitanalyse für die Motorsteuergeräte-Software, erläutert werden. Aus der globalen Laufzeitproblematik wird die Forderung nach einer lokalen Laufzeitanalyse für eine einzelne Fahrzeugfunktion als Software-Komponente abgeleitet.

2.1 Echtzeitsystem

Wie in der Einführung bereits erwähnt wurde, unterliegen Echtzeitsysteme zeitlichen Anforderungen, diese werden durch ihre Anwendung vorgegeben. Der Begriff Echtzeit (engl. real time) wird in der DIN 44300 [18] als Realzeitverarbeitung folgendermaßen definiert:

„Eine Verarbeitungsart, bei der Programme zur Verarbeitung anfallender Daten ständig ablaufbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen."

Ein Echtzeitsystem ist ein Rechensystem, dessen Korrektheit von der logischen Korrektheit der Ergebnisse und der Einhaltung vorgegebener Zeitbedingungen abhängt. Die Nichteinhaltung der vorgegebenen Zeitforderungen ist gleichbedeutend mit dem Versagen des Systems.
Eine allgemeine Einteilung von Echtzeitanforderungen erfolgt in harte und weiche Echtzeitanforderungen, die Literatur ist bei dieser Einteilung nicht eindeutig. Diese Arbeit orientiert sich an den Definitionen und Festlegungen in [1, 25, 39]. Die formale Festlegung von harten Zeitanforderungen betrachtet dabei die Zeitparameter:

- Bereitzeitpunkt t_R (engl. ready time): ist der Zeitpunkt, zu dem die Aufgabe beginnt,
- Fristzeitpunkt t_D (engl. deadline): ist der Zeitpunkt, zu dem die Aufgabe erledigt sein muss, und
- Ausführungszeit Δt_E (engl. execution time): ist die Zeitspanne, die die Ausführung der Aufgabe in Anspruch nehmen darf, um der Forderung nach Rechtzeitigkeit zu genügen.

$$t_R + \Delta t_E \leq t_D \tag{2.1}$$

Die wissenschaftliche Auseinandersetzung mit dem Thema Echtzeitfähigkeit findet vor allem mit der globalen Zeitanalyse statt. Hier werden Betrachtungen vorgenommen, die konkurrierende Aufgaben unterschiedlicher Wichtigkeit betrachten. Dabei werden die globalen Zeitbe-

dingungen analysiert, wie beispielsweise die Berücksichtigung von Aufgaben mit Vorrang. Dies wird durch einen weiteren Zeitparameter berücksichtigt:

- Ausführungszeit einer Aufgabe mit Vorrang $\Delta t_{E,V}$: ist die maximale Zeitspanne, die die Ausführung einer Aufgabe mit Vorrang in Anspruch nehmen darf, um der Forderung nach Rechtzeitigkeit zu genügen.

Die Auswirkungen der Dauer $\Delta t_{E,V}$ muss in Gl. 2.1 übertragen werden (s. Gl. 2.2), die Berücksichtigung von mehreren Aufgaben mit Vorrang erfolgt z. B. in [4] und [41].

$$t_R + \Delta t_E + \Delta t_{E,V} \leq t_D \tag{2.2}$$

Weiche Echtzeitsysteme erfüllen diese Anforderungen nicht, d. h., es kann nicht sichergestellt werden, dass unter allen Einsatzbedingungen die Forderung nach Rechtzeitigkeit erfüllt ist. Harte Echtzeitsysteme müssen diese Anforderungen erfüllen. Die Anforderung [40], die Korrektheit und Rechtzeitigkeit des Verhaltens eines harten Echtzeitsystems unter allen Einsatzbedingungen im Voraus eindeutig zu bestimmen, entspricht meist nicht der Realität. Eine Verifikation der vorgegebenen Zeitbedingung wird meist nur dann gefordert, wenn ein nicht rechtzeitiges Ergebnis zu einer Schädigung von Menschen oder einer Zerstörung von Einrichtungen bzw. zu finanziellen Verlusten führt.

2.1.1 Eingebettetes System

Echtzeitsysteme sind vielfach als eingebettete Systeme realisiert. Ein Rechnersystem bestehend aus Hard- und Software, das in ein umgebendes technisches System eingebettet ist, wird als eingebettetes System bezeichnet. Es wird meist als steuerndes System zur Regelung, Steuerung und Überwachung eines gesteuerten technischen Systems benutzt [24], wie zum Beispiel einem Fahrzeugmotor. Der typische Aufbau ist in Abb. 2.1 (a) dargestellt.

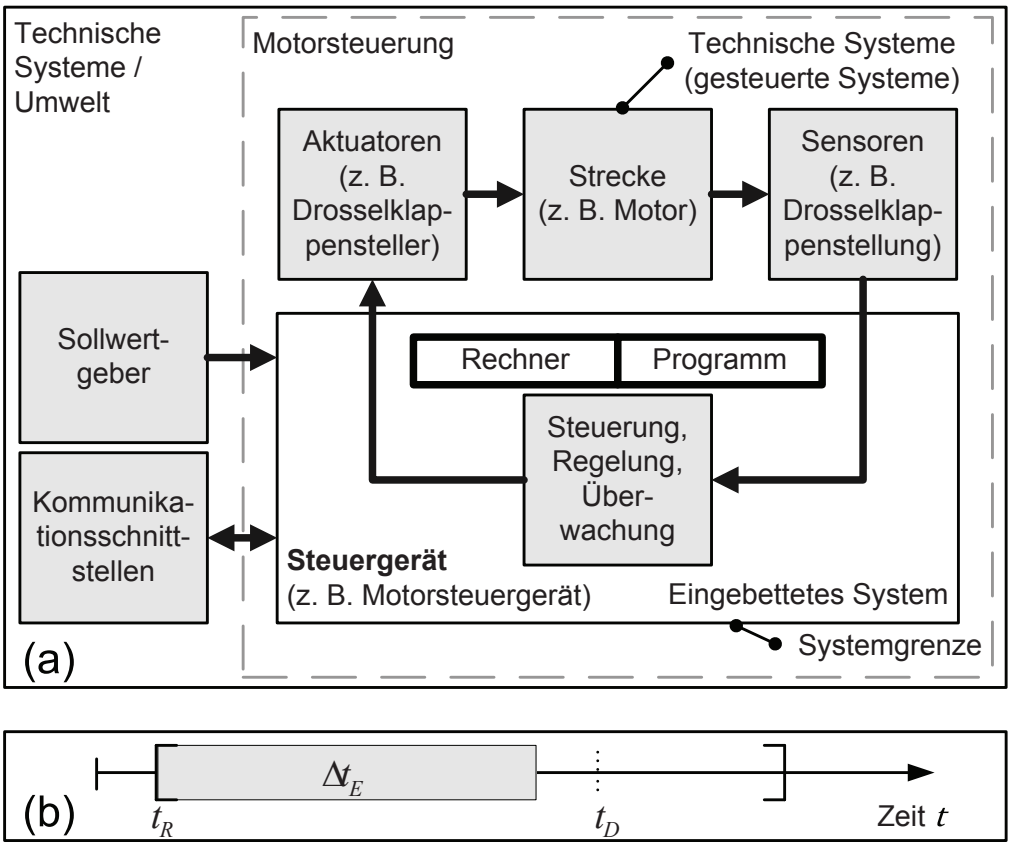


Abb. 2.1: (a) eingebettetes Echtzeitsystem - Motorsteuerung (b) Rechenzeitbedarf

Ein solches eingebettetes System basiert auf einer speziellen Hardwareplattform, die über beschränkte Ressourcen verfügt. In Abb. 2.1 (b) ist die Zeit als beschränkte Ressource als abgeschlossenes Intervall durch eckige Klammern gekennzeichnet. Die Funktionalität wird auf dieser Plattform als Software implementiert, in Abb. 2.1 (b) ist der Rechenzeitbedarf für ein Programm mit $\Delta t_E < t_D - t_R$ abgebildet.

Schnittstellen verbinden über die Systemgrenzen hinaus das eingebettete System mit der Umwelt bzw. mit dem gesteuerten technischen System. Das einzelne Steuerungssystem setzt sich zusammen aus den Sensoren zur Datenaufnahme, dem Steuergerät und den Aktuatoren zur Beeinflussung der Regelstrecke und Kommunikationsschnittstellen zum Datenaustausch mit anderen (eingebetteten) Systemen in der Umgebung. Die Sensoren bereiten beliebige physikalische Messsignale (z. B. Zustandssignale des gesteuerten Systems) als elektrische Signale auf und stellen diese dem Steuergerät zur Verfügung. Im Steuergerät werden diese Eingangssignale verarbeitet und entsprechend der Logik des Steuerungssystems daraus elektrische Ausgangssignale berechnet.

Der Aktuator wandelt und verstärkt diese elektrischen Ausgangssignale in andere physikalische Steuersignale, die dann das gesteuerte System beeinflussen. Die Hardwareplattform eines Steuergerätes basiert oftmals auf einem Mikrocontroller [9]. Dieser Mikrocontroller ist unter anderem bereits mit einem integrierten Prozessor (zentrale Recheneinheit), einem Bussystem, Speicher (RAM, ROM/EPROM, Flash/EEPROM), analogen und digitalen Ein- und Ausgängen sowie meist einem externen Bus zur Kommunikation mit anderen Bausteinen ausgestattet. Abgesehen vom Mikrocontroller beherbergt die Hardwareplattform oftmals noch eine Spannungsaufbereitung, Eingangsschaltungen für die Sensorik, Treiber für die angeschlossenen Aktuatoren (z. B. Steller für Drosselklappe [77]) und je nach Anwendungsfall weitere Speicherbausteine (externes RAM/Flash), Kommunikationstreiber (z. B. Controller für CAN [77]) oder beliebige andere Bauelemente. Neben den Sensoren gibt es noch die Sollwertgeber (z. B. Fahrpedalsensor zur Erfassung der aktuellen Gaspedalstellung [77]), die dem Fahrer als (Eingabe-) Schnittstellen für das Steuergerät dienen. In diesem Sinn sind die Sollwertgeber für das Steuergerät Sensoren.

2.1.2 Eingebettetes Software-System

In Anlehnung an AUTOSAR (Automotive Open System Architecture) [35] kann die Software-Architektur eines Steuergerätes vereinfacht wie in Abb. 2.2 vorgestellt werden. Im AUTOSAR Konsortium haben sich Automobilhersteller und Zulieferer zusammengeschlossen mit dem Ziel der Definition eines offenen Standards für die Architektur von Elektronikkomponenten im Automobilbereich. AUTOSAR setzt dabei auf bestehenden Standards auf, die im Folgenden eingeführt werden.

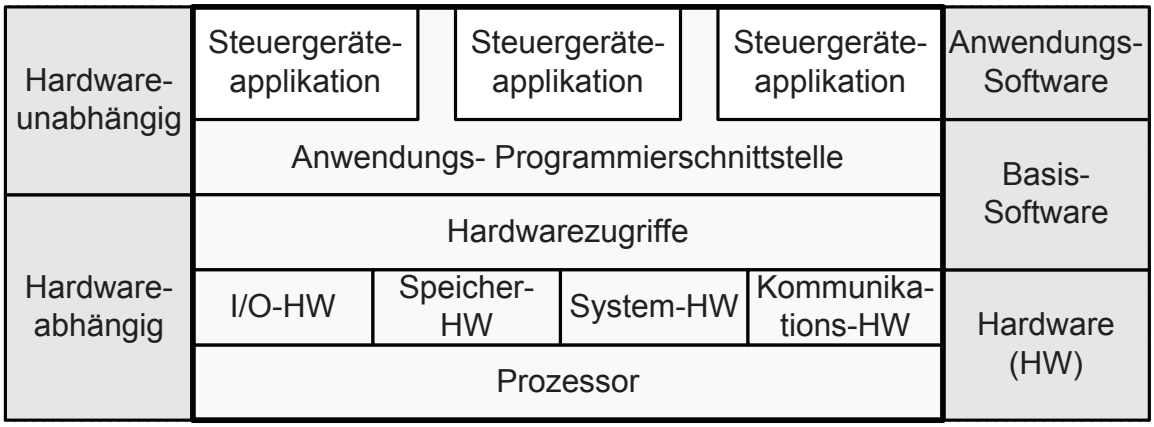


Abb. 2.2 Steuergerätesoftware-Architektur [24]

Software-Architektur eines Steuergerätes

Die Software-Architektur eines Steuergerätes kann in die Basissoftware bzw. Systemsoftware und Anwendungssoftware bzw. Applikationssoftware eingeteilt werden. Der Basissoftware kommt dabei die Aufgabe zu, die Anwendungssoftware zu steuern, zu verwalten und zu unterstützen. Sie bildet damit die Schnittstelle zwischen Hardware und Software und kann grob in:

- Betriebssystem und
- Treibersoftware für Hardwarezugriffe unterteilt werden.

Die Anwendungssoftware stellt eine Menge von auszuführenden Programmen dar, die in die typischen Komponenten:

- Funktionssoftware für Regel- und Steuerungsaufgaben sowie
- Überwachungs- und Diagnosesoftware für Fehlererkennung und Fehlerbehandlung eingeteilt ist.

Überwachungs- und Diagnosefunktionen sind auch in der Basissoftware zu finden, um beispielsweise die Hardware zu überwachen. Die Anwendungsschnittstelle[1] dient zur Verbindung der einzelnen Applikation bzw. Anwendungen miteinander und zur Verbindung mit der Basissoftware und damit zur Hardware.

Mit der Einführung eines zukünftigen Standards durch AUTOSAR sollen die Entwicklungskosten und Risiken für Elektronikkomponenten im Automobilbereich reduziert werden. Dafür werden Standards festgelegt, die grundlegende Systemfunktionen, Basisfunktionalitäten und Funktionsschnittstellen normieren. Dabei baut AUTOSAR auf den folgenden Standards auf, wie sie in der aktuellen Steuergerätegeneration zu finden sind.

Standards und Standardisierungsgremien

Die wichtigsten Standards bzw. Standardisierungsgremien sind (wie in [41] zusammengestellt wurde):

1. OSEK[2]:
OSEK vereinheitlicht die Schnittstellen für Echtzeitbetriebssysteme (OSEK-OS), Kommunikationsdienste (OSEK-COM) und Netzwerkdienste (OSEK-NM) für Anwendungen von Steuergeräten im Kraftfahrzeug [6].

2. ASAM[3]:
Der ASAM-AE (Automotive Electronics) Standard mit den einzelnen Standards MCD (Measurement Calibration and Diagnosis) 1, 2 und 3 stellt Schnittstellen zur Beschreibung und Integration von Steuer- und Regelsystemen (Mess-, Kalibrier- und Diagnoseschnittstellen) dar. ASAM hat diese vom ASAP (Arbeitskreis zur Standardisierung von Applikationssystemen) übernommen. MCD 2 definiert zum Beispiel ein Dateiformat, das alle Applikationsparameter und Messgrößen eines Steuergerätes beschreibt. Eine Datei mit diesen Inhalten wird umgangssprachlich ASAP-Datei genannt [42]. Weitere in den ASAM übernommene Standards kommen aus den MSR (Manufacturer Supplier Relationship) Arbeitskreisen. Hier wurden zum Beispiel Formate auf der Basis von XML (Extended Markup Language) definiert, die zur Beschreibung von Software (MSRSW-DTD) oder zur Dokumentation der Software (MSR-MEDOC) dienen.

[1] Middleware, zur Unterstützung der Kommunikation zwischen entkoppelten Software-Komponenten
[2] Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
[3] Association for Standardization of Automation and Measuring Systems

3. AUTOSAR:
Beschäftigt sich mit dem Aufbau der Kfz-Software allgemein und setzt dabei u. a. auf bestehende Standards von MSR und ASAM. Die derzeitigen Steuergerätegenerationen basieren auf älteren Standards als AUTOSAR, die auf der Herstellerinitiative Software (HIS) [41] und OSEK aufbauen. HIS ist ein Zusammenschluss von Fahrzeugherstellern mit dem Ziel der Vereinheitlichung von Anforderungen an Komponenten und Prozesse. In fünf HIS-Arbeitskreisen (Prozess-Assessment, Software-Test, Standard-Software, Flashbare Software sowie Simulation und Tools) werden die Standards für die Erstellung wieder verwendbarer, hochqualitativer Steuergeräte-Software definiert.

Echtzeit-Betriebssystem

Die Ausführung der verschiedenen Algorithmen und Berechnungen eines eingebetteten Echtzeitsystems sowie die Zuteilung der Ressourcen eines Prozessors werden durch das Echtzeit-Betriebssystem gesteuert. Die nachfolgende Betrachtung erfolgt für ein Einzelprozessorsystem, d. h. nur eine Anwendung kann zu einem Zeitpunkt ausgeführt werden. Die Echtzeitplanung (engl. real-time scheduling) ist die Aufgabe, den Prozessor unter Einhaltung der Zeitbedingungen den Anwendungen zuzuteilen. Nach [26] ist der Kern eines exemplarischen Betriebssystems ein prioritätsbasiertes, *kooperatives* und *preemptives* Scheduling, um so die hohe Anzahl der parallelen Anforderungen an das eingebettete System zu bedienen. In dieser Multitasking-Umgebung wird die Ausführung von Tasks durch diesen Scheduler gesteuert. Durch die potentiell parallel abzuarbeitenden Aufgaben, im Folgenden als Task bezeichnet, wird eine bessere Auslastung des Prozessors erreicht.

Definition 2.1 Task: Ein Task $\boldsymbol{\tau_i}$ besteht aus einer Folge von Prozessen $\boldsymbol{Pr}=\{\boldsymbol{pr_i}:i=1...N\}$ mit identischen Echtzeitanforderungen, die nach dem Start des Tasks in der vorgegebenen Reihenfolge sequentiell abgearbeitet werden. Tasks sind entweder aperiodisch und werden in unregelmäßigen Abständen aktiviert oder periodisch und werden in regelmäßigen zeitlichen Abständen aktiviert.

Für die Menge der Task $\boldsymbol{K}$ gilt $\boldsymbol{K}=\{\boldsymbol{\tau_i}:i=1...N\}$.

Anmerkung: Der Begriff Prozess wird hier nach dem OSEK-Standard angewandt und bedeutet nicht wie in der Literatur häufig zu finden eine parallel abzuarbeitende Aufgabe. Diese Aufgabe erfüllt nach OSEK der Task.

Bestimmte Teile eines Regelalgorithmus von Fahrzeug-Funktionen, die mit einer vorgegebenen Frequenz (periodisch) oder als Reaktion auf eine externe Unterbrechung (aperiodisch) auszuführen sind, werden als Prozess ausgeführt. Im Allgemeinen wird auf einem Steuerungssystem eine Vielzahl von Algorithmen zur Ausführung gebracht, damit ist auch die Zahl der Prozesse sehr hoch. Viele dieser Prozesse mit demselben dynamischen Verhalten lassen sich zu Tasks zusammenfassen. Dadurch wird der Verwaltungsaufwand des Betriebssystems verringert und das dynamische Verhalten der Anwendung wird strukturiert. Zu den zeitlichen Bedingungen bzw. Anforderungen, denen ein Echtzeittask unterliegt, gehören:

- diverse Zeitparameter (engl. timing constraints), die das zeitliche Verhalten des Tasks bestimmen,
- Vorrangrelationen (engl. precedence relation), die die Abhängigkeiten zwischen verschiedenen Tasks definieren sowie
- der exklusive Zugriff auf gemeinsam benutzte Betriebsmittel bzw. Ressourcen (engl. mutual exclusion constraints of shared resources).

Ein Task wird auf einem Prozessor sequentiell zur Ausführung gebracht. Die Ausführung mehrerer Tasks auf einem Prozessor macht eine zeitliche Aufteilung der Ressourcen des Pro-

zessors auf die unterschiedlichen Tasks erforderlich. Dabei muss zu gewissen Zeitpunkten zwischen den Tasks umgeschaltet werden, das wird als Kontextwechsel (engl. context switch) bezeichnet. Dies kann prioritätsbasiert erfolgen, die Priorität des jeweiligen Tasks legt die Wichtung fest, mit der er zur Ausführung gebracht wird. Zudem wird mit der Priorität festgelegt, in wieweit sich die Tasks untereinander unterbrechen können. Ein laufender Task kann nur von einem Task mit höherer Priorität unterbrochen werden. Die verschiedenen Zustände von Tasks sowie die Ereignisse, welche eine Taskumschaltung einleiten, und die Strategien einer Taskumschaltung in einem OSEK konformen Echtzeitsystem sind unter [6] einzusehen. Eine einfache Einteilung der Zustände eines Tasks sieht wie folgt aus:

- aktiv (engl. active), wenn der Task prinzipiell ausgeführt werden kann, unabhängig von der tatsächlichen Verfügbarkeit eines freien Prozessors, und
- blockiert (engl. blocked), wenn der Task auf das Eintreten einer Bedingung wartet (bspw. das Freiwerden eines Betriebsmittels), bevor er zur Ausführung gebracht werden kann.

Die aktiven Tasks lassen sich weiterhin unterteilen in:

- bereit (engl. ready), wenn der Task auf die Ausführung auf einem Prozessor wartet, und
- laufend (engl. running), wenn er auf einem Prozessor ausgeführt wird.

Die Tasks, die bereit zur Ausführung sind, werden in einer Warteschlange vermerkt, der Bereitschlange (engl. ready queue).

Ein Task unterliegt den folgenden Zeitparametern:

- Startzeitpunkt t_S (engl. start time): mit diesem Startzeitpunkt beginnt die Ausführung des Tasks durch den Prozessor und
- Abschlusszeitpunkt t_C (engl. completion time): mit diesem Abschlusszeitpunkt endet die Ausführung des Tasks durch den Prozessor.

Der allgemeine Fall für die Beziehungen zwischen Startzeitpunkt, Ausführungszeit und Abschlusszeitpunkt einer Task wird durch die Ungleichung (2.3) beschrieben.

$$t_S + \Delta t_E \leq t_C \tag{2.3}$$

Für den speziellen Fall eines nicht unterbrechbaren Tasks gilt Gl. (2.4).

$$t_S + \Delta t_E = t_C \tag{2.4}$$

Die Ausführungszeit für harte Echtzeitsysteme für den allgemeinen und speziellen Fall muss die Bedingung (2.5) für die Zeitintervalle erfüllen.

$$[t_S, t_C] \subseteq [t_R, t_D] \tag{2.5}$$

Zur Veranschaulichung werden die Beziehungen von zeitlich überlappenden Taskausführungen, also Nebenläufigkeit, exemplarisch in Abb. 2.3 dargestellt. Zur Vereinfachung erfolgt mit der Aktivierung des Tasks auch der Start des Tasks, d. h. die Ausführung durch den Prozessor. Die Prozesse eines Tasks werden in einer festen sequentiellen Folge ausgeführt und mit pr_1 bis pr_6 bezeichnet. Ein Prozess wird aus einer Folge von Anweisungen aufgebaut, je nach Scheduling-Strategie bzw. Planungsstrategie können diese Anweisungen unterbrochen werden. Die wesentlichen Zusammenhänge, ob die Zuteilung des Prozessors eines bevorrechtigten Tasks (engl. preemptive) ohne oder mit Verdrängung (engl. suspension) des Tasks erfolgt, der gerade ausgeführt wird, sind in Abb. 2.3 dargestellt.

Beim *preemptiven* Scheduling wird der laufende Prozess (pr_2 im Task $\boldsymbol{\tau_1}$) direkt unterbrochen, sobald ein Task (Task $\boldsymbol{\tau_2}$) mit einer höheren Priorität aktiviert wird. Nachdem der unterbrechende Task (Task $\boldsymbol{\tau_2}$) abgeschlossen ist, wird der Prozess (pr_2 im Task $\boldsymbol{\tau_1}$) fortgesetzt.

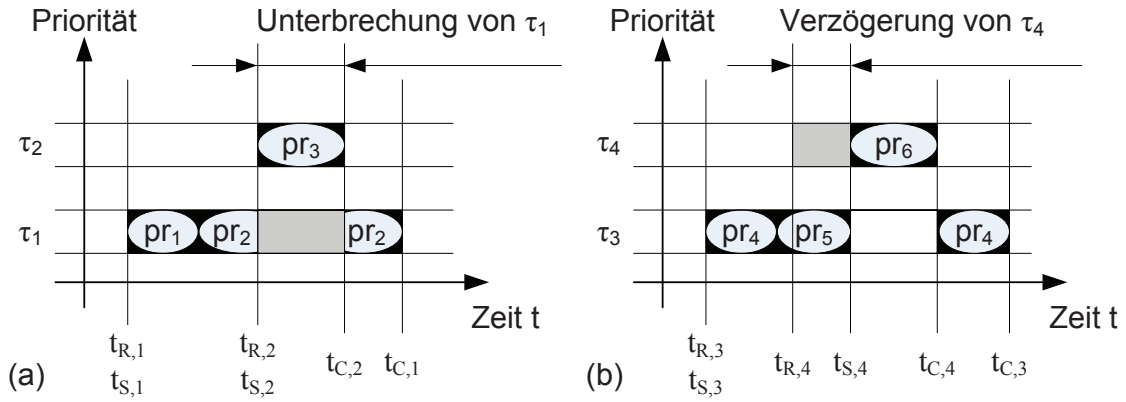


Abb. 2.3: OSEK-OS [5] (a) *preemptives* Scheduling (b) *kooperatives* Scheduling

Beim *kooperativen* Scheduling wird der laufende Prozess (pr_5 im Task τ_3) nicht unterbrochen, wenn ein Task (Task τ_4) mit einer höheren Priorität aktiviert wird. Ein Task (Task τ_3) kann nur an einer Prozessgrenze unterbrochen werden, das heißt nur dann, wenn sich kein Prozess in der Ausführung befindet. Nachdem der unterbrechende Task (Task τ_4) abgeschlossen ist, wird der unterbrochene Task (Task τ_3) fortgesetzt.

Motorsteuerung

Zur Verdeutlichung der Vorgänge in einer Motorsteuerung soll die nachfolgende Erläuterung dienen. Im Wesentlichen muss zwischen drehzahlsynchronen (aperiodischen) und zeitsynchronen (periodischen) Echtzeitanforderungen bei Motorsteuergerätefunktionen unterschieden werden. Drehzahlsynchrone Aufgaben, wie die Zündwinkel-Sollwert-Berechnung zur Zündwinkelverstellung [77], werden im „drehzahlsynchronen Task“ mit variabler Abtastrate entsprechend der Motordrehzahl gerechnet. Typische Abtastraten bei hohen Motordrehzahlen liegen bei 1-2 ms, dies entspricht dem *preemptiven* Task τ_2 in Abb. 2.3.a. Mit festen Abtastraten kommen dagegen Funktionen zur Lageregelung wie die Sollwert-Berechnung-Frischgasfüllung (elektrisch gesteuerte Drosselklappe [77]) aus, diese „zeitsynchronen Tasks“ haben typische Abtastraten von 10 ms oder 20 ms, in Abhängigkeit von den Anforderungen an das dynamische Verhalten. Weniger dynamische Anforderungen stellen dagegen temperaturgeführte Funktionen, hier sind „zeitsynchronen Tasks“ mit 100 ms und 1000 ms Abtastraten zu finden. Abbildung 2.3.b stellt ein Beispiel für einen kooperativen Task τ_4 mit 20 ms und τ_3 mit 100 ms dar.

2.2 Echtzeitanalyse

Bei der Laufzeitanalyse eines Echtzeitsystems, wie es in den vorangegangenen Abschnitten vorgestellt wurde, muss untersucht werden, ob alle zeitlichen Anforderungen der Spezifikation entsprechend umgesetzt wurden. Das bedeutet, dass alle spezifizierten Zeitschranken (s. Abb. 2.4) eingehalten werden, um damit die geforderte Rechtzeitigkeit des Ergebnisses eines Echtzeitsystems nach Abschn. 2.1 zu garantieren. Für die Rechtzeitigkeit eines Ergebnisses wurde ein spätest möglicher Zeitpunkt t_D (Frist, engl. deadline) eingeführt, bis zu dem die Antwort erwartet wird.

Definition 2.2 (Task) Antwortzeit [27]**:** Die Antwortzeit Δt_A ist die Zeitspanne vom Bereitwerden t_R bis zur Vorlage des Ergebnisses (Antwort) t_C.

Die Antwortzeit (engl. response time) stellt die gesamte Fertigstellungsdauer dar, das heißt einschließlich der Speicherzugriffe, E/A-Aktivitäten, Betriebssystem-Overhead und der Bearbeitung von weiteren unterbrechenden Tasks in einer Multitasking-Umgebung. Eine Ver-

letzung des Deadline-Zeitpunktes t_D bedeutet bei dem hier betrachteten harten Echtzeitsystem ein Versagen des Systems. Der Zeitpunkt des Bereitwerdens t_R und der spätest mögliche Zeitpunkt t_D der Antwort sind die Bedingungen, die die maximal zulässige Antwortzeit $\Delta t_{A\max}$ festlegen.

2.2.1 Tasklaufzeit

Definition 2.3 Tasklaufzeit: Die Tasklaufzeit $\Delta t_{E,i}^{j}$ ist die Zeitspanne vom Startzeitpunkt $t_{S,i}^{j}$ bis zum Abschlusszeitpunkt $t_{C,i}^{j}$ des Tasks *i*, für die *j-te* Ausführung auf einem Prozessor im unterbrechungsfreien Betrieb.

Die Tasklaufzeit stellt die gesamte Ausführungszeit (engl. execution time) von beliebigen in einem Task organisierten Programmen dar, die auf einem Computersystem zur Abarbeitung der Programme benötigt wird. Erfolgt eine Abarbeitung der Programme eines Tasks ohne Unterbrechung, wie in Abb. 2.4, ist die Tasklaufzeit die Zeitspanne zwischen Start und Abschluss der Ausführung des Tasks. Kommt es zu einer Unterbrechnung der Abarbeitung, ist die Summe der Abarbeitungszeiten zwischen Start und Abschluss der Ausführung des Tasks die Tasklaufzeit. Die Ausführungszeit eines Tasks ist abhängig von einer Vielzahl von Einflussgrößen, die sowohl Software als auch Hardware bedingt sind. Einen wesentlichen Einfluss haben dabei die Eingangsdaten, das bedeutet, dass die einzelnen Programme unterschiedliche Programmpfade der Programmausführung haben und dass diese von den Eingaben abhängen. In Kapitel 3 werden die laufzeitbestimmenden Größen identifiziert und diskutiert. Diese Abhängigkeiten führen zu unterschiedlich langen Tasklaufzeiten und machen eine Bestimmung der maximalen Ausführungszeit, das heißt, eine Worst-Case-Abschätzung für die Ausführungszeit jedes Tasks, notwendig.

Definition 2.4 Tasklaufzeitgrenzen: Die obere und untere Laufzeitgrenze eines Tasks $(\Delta t_{E\min,i}, \Delta t_{E\max,i})$ sind die minimale und maximale von $\Delta t_{E,i}$, $\Delta t_{E,i} \in \Delta T_i$, d. h. $\Delta t_{E\min,i} \le \Delta t_{E,i} \le \Delta t_{E\max,i}$. Dabei ist ΔT_i die Menge aller Laufzeiten des Tasks i.

Die Laufzeitgrenzen beziehen sich dabei immer auf eine definierte Hardwareumgebung. Allgemein wird die Zeitspanne $\Delta t_{E\max,i}$ als die längstmögliche Rechenzeit des Tasks mit Worst-Case Execution Time (*WCET*) $\Delta t_{WCET,i}$ bezeichnet und $\Delta t_{E\min,i}$ als die kürzestmögliche Rechenzeit mit Best Case Execution Time (*BCET*) $\Delta t_{BCET,i}$. Die maximal zulässige Antwortzeit $\Delta t_{A\max,i}$ legt ein Zeitfenster für die Ausführung des Tasks fest, in dem eine Antwort (Taskantwort) erwartet wird. Das Zeitfenster wird aus dem Aktivierungszeitpunkt t_R, d. h. aus dem Zeitpunkt zu dem die Ausführung des Task angestoßen oder freigegeben wird (siehe [26]), und der Zeitschranke t_D gebildet. Die Zeitschranke t_D ist der Zeitpunkt, zu dem die Ausführung des Tasks spätestens abgeschlossen sein muss. Für die Wahrung der Echtzeitfähigkeit muss die WCET eines Tasks innerhalb des Zeitfensters liegen, das für die Ausführung des Tasks vorgesehen ist.

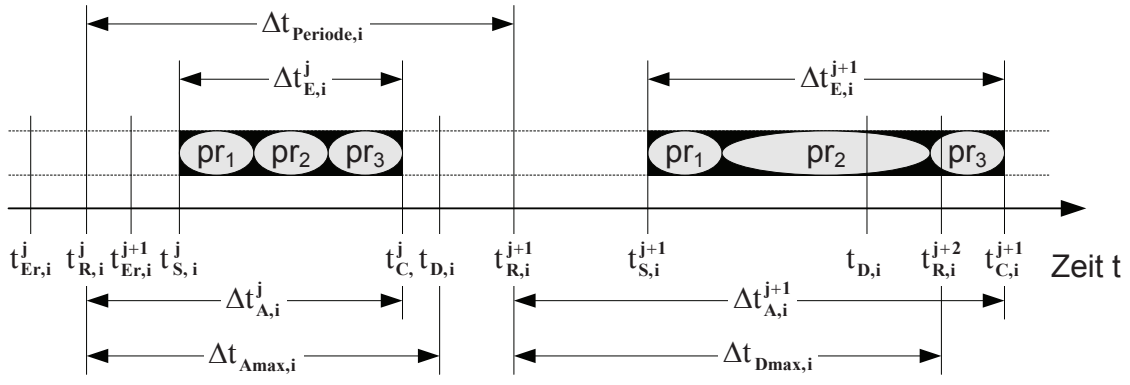


Abb. 2.4 Festlegung von Echtzeitanforderungen [25]

Die Länge des Zeitfensters wird außerdem durch die Zeitspanne zwischen zwei Aktivierungen eines Tasks bestimmt. Diese Zeitspanne wird allgemein als Rechenraster oder Periode des Tasks bezeichnet. Für periodische Task kann folgende Verallgemeinerung getroffen werden:

- Periode $\Delta t_{Periode,i}$: ist die Zeitspanne zwischen zwei aufeinanderfolgenden Aktivierungen des Tasks.

$$t_{R,i}^{j} = (j-1)\Delta t_{Periode} + t_{R,i}^{1} \qquad j \geq 1 \tag{2.6}$$

Bei einer statischen Zuteilung der Prozessor-Ressourcen können alle Zuteilungsentscheidungen vor der Ausführung des Programms getroffen werden. Ausgehend von der ersten Aktivierung des Tasks $t_{R,i}^{1}$ sind damit alle weiteren Aktivierungen nach Gl. (2.6) vorgegeben. Dies bedeutet, im ungünstigsten Fall liegt zwischen dem Ereignis t_{Er} und der Aktivierung des Tasks t_R ein Rechenraster, wenn eine Übernahme der Ereignisdaten nur zum Zeitpunkt der Aktivierung des Tasks erfolgt. Dies muss bei der Definition des Fristzeitpunktes und damit bei der Periodendauer periodischer Tasks berücksichtigt werden. Das Ende der Periode setzt somit auch den Fristzeitpunkt nach:

$$\begin{aligned} t_{D,i}^{j} &= j\Delta t_{Periode,i} + t_{R,i}^{1} \qquad j \geq 1 \\ &= t_{R,i}^{j+1} \end{aligned} \tag{2.7}$$

Um schneller auf Ereignisse reagieren zu können, werden dynamische Zuteilungsstrategien eingesetzt, mit denen die Zeit zwischen dem Ereignis t_{Er} und der Aktivierung des Tasks t_R verkürzt werden kann. Weiterhin müssen bei der Festlegung des Zeitfensters die Zeiten für die Abarbeitung weiterer Tasks (Tasks in anderen Zeitrastern) berücksichtigt werden.

Motorsteuerung

In der realen Multitasking-Umgebung eines Motorsteuergerätes kommt dem Systemintegrator die Aufgabe des Schedulability-Tests (bzw. der globalen Echtzeitanalyse) zu, bei dem die Einhaltung der Echtzeitanforderungen geprüft werden. Nach [23] muss hierbei analysiert werden, ob die Tasks mit ihren berechneten Abarbeitungszeiten, Precedence Constraints und Ressourcenanforderungen auf dem gegebenen Rechnersystem so abgearbeitet werden können, dass das geforderte Zeitverhalten garantiert werden kann. Hierfür benötigt der Systemintegrator die maximale Laufzeit von jedem Task (Worst-Case Execution Time $\Delta t_{WCET,i}$).

2.2.2 Prozesslaufzeit

Im vorangegangenen Abschnitt wurde für einen Task ein Zeitfenster $\Delta t_{A\max}$ eingeführt, das mit der Aktivierung des Tasks t_R beginnt und mit einem harten Fristzeitpunkt t_D endet. Die Festlegung dieses Task-Zeitfensters genügt jedoch nicht, wenn die Laufzeitinformationen von einzelnen Software-Komponenten dem Systemintegrator nicht offen liegen. Dies ist der Fall, wenn die Software-Komponente aus einer verteilten Entwicklung zwischen Fahrzeughersteller (als Software-Lieferant) und Systemlieferant hervorgeht. Dem Systemintegrator fehlt für eine vollständige Beschreibung der *WCET* des Tasks die *WCET* der Software-Komponente. Anhand der Abb. 2.4 soll dieser Sachverhalt noch einmal verdeutlicht werden. Das vorgegebene Zeitfenster wird bei der zweiten Aktivierung des Tasks $t_{R,i}^{j+1}$ verfehlt, das führt zu einer Antwort $t_{C,i}^{j+1}$, die nach der harten Frist $t_{D,i}$ des Echtzeitsystems liegt. Dies resultiert aus der im Vergleich zur Taskausführung j bei einer in $j+1$ längeren Laufzeit des Prozesses pr_2. Da sich ein Task aus einer Sequenz von Prozessen, die den Ausführungscode der Programme enthalten, zusammensetzt, muss das Zeitfenster für den Task auf die Prozesse übertragen werden. Hierfür wird ein Prozess-Container definiert, der die Komplexität, das heißt den Zeit- und Speicheraufwand für eine Berechnung einer Funktion in Form eines Programms festlegt.

Nachfolgend wird ein Prozess-Container betrachtet, der lediglich auf sein Zeitverhalten reduziert ist, d. h. auf die maximal zulässige Prozesslaufzeit. Eine minimal zulässige Prozesslaufzeit wird nicht definiert, es wird lediglich das Bedienen des Prozessaufrufes im Prozess-Container verlangt.

Anmerkung: Die Notation folgt im Weiteren der des bereits eingeführten Tasks, jedoch wird zur Unterscheidung zwischen Taskzeitparameter und Prozesszeitparameter ein Pr für Prozess angefügt.

In Anlehnung an [2] wird die Prozesslaufzeit *t* wie folgt definiert:

Definition 2.5 Prozesslaufzeit: Die Prozesslaufzeit $\Delta t_{\mathrm{Pr}E}$ ist die Zeit, die bei gegebener Hardware und gegebenen Eingabedaten für die Erfüllung der Funktionalität der Fahrzeugfunktion gemäß ihrer internen Logik aufgewendet wird.

Die Prozesslaufzeit stellt dabei die Ausführungszeit eines Programms ohne die Zeit für die externe Kommunikation dar. Die Kommunikation zwischen Fahrzeugfunktion (Prozessen), dem technischen System (Umwelt) oder auch zu anderen Steuergeräten erfolgt ausschließlich über eine Anwendungsschnittstelle.

Anwendungsschnittstelle

Die Anwendungsschnittstelle, wie sie bereits im Abschnitt „Eingebettetes Software-System“ eingeführt wurde, wird auch als Software-Adapter bezeichnet und ist zwischen dem Automobilhersteller und dem Systemliefereranten definiert. Der Adapter beinhaltet die notwendigen Informationen zu Signalen wie Name, Quantisierung, Richtung (Eingangs- oder Ausgangsgröße), Datentypen, min./max. Werte, Initialisierungswerte und Rechenraster. Die Kommunikation oder der Zugriff auf andere Komponenten, Treiber (HW-Interfaces) oder zentrale Funktionen erfolgt ausschließlich über diese Schicht. Der Software-Adapter liegt in der Hand des Systemlieferanten. Die Funktionalität der Schnittstelle wird dabei auf mehrere Prozesse verteilt, wie in Abb. 2.5 dargestellt:

- Der Prozess A1 ist die Eingangsschnittstelle zur Konvertierung von Signalen der Systemsoftware in Eingangssignale für die Software-Komponente.
- Der Prozess A2 ist die Ausgangsschnittstelle zur Konvertierung von Ausgangssignalen der Software-Komponente in Signale der Systemsoftware.

Als Beispiel für Signale, die über diese Schnittstellen übergeben werden, seien hier die Motordrehzahl als Eingangssignal und als Ausgangssignal das Ansteuersignal für die Soll-Position der Drosselklappe genannt.

Die maximal zulässige Prozesslaufzeit $\max(\Delta t_{\mathrm{Pr}E})$ wird durch den zur Verfügung stehenden Prozess-Container festgelegt. Für die Ausführung der Fahrzeugfunktion steht damit ein definiertes Prozesszeitfenster $\Delta t_{\mathrm{Pr}A\max}$ fest, in dem eine Antwort erwartet wird. Das Prozesszeitfenster wird aus dem Startzeitpunkt $t_{\mathrm{Pr}S}$, d. h. aus dem Zeitpunkt, zu dem die Ausführung des Prozesses beginnt und der Prozesszeitschranke $t_{\mathrm{Pr}D}$ festgelegt. Die Prozesszeitschranke $t_{\mathrm{Pr}D}$ ist der Zeitpunkt, zu dem die Ausführung des Prozesses (Prozessende $t_{\mathrm{Pr}C}$) spätestens abgeschlossen sein muss. Für die Wahrung der Echtzeitfähigkeit muss die *WCET* eines Prozesses $\Delta t_{\mathrm{Pr}WECT}$ innerhalb des Prozesszeitfensters $\Delta t_{\mathrm{Pr}A\max}$ liegen, das für die Ausführung des Prozesses spezifiziert ist.

Definition 2.6 Prozesslaufzeitgrenzen: Die obere und untere Laufzeitgrenze eines Prozesses $(\Delta t_{\mathrm{Pr}E\min,i}, \Delta t_{\mathrm{Pr}E\max,i})$ sind die minimale und maximale von $\Delta t_{\mathrm{Pr}E,i}$, $\Delta t_{\mathrm{Pr}E,i} \in \Delta T_{\mathrm{Pr}i}$, d. h. $\Delta t_{\mathrm{Pr}E\min,i} \le \Delta t_{\mathrm{Pr}E,i} \le \Delta t_{\mathrm{Pr}E\max,i}$. Dabei ist $\Delta T_{\mathrm{Pr}i}$ die Menge aller Laufzeiten des Prozesses i.

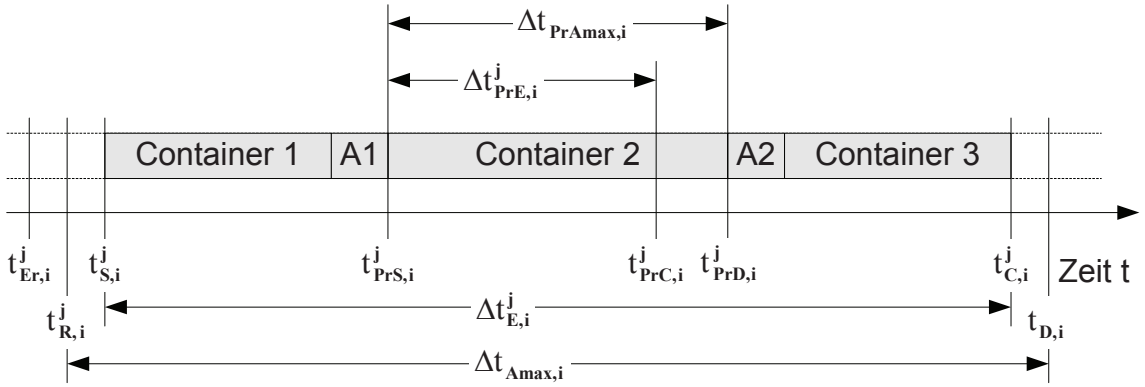


Abb. 2.5: Prozessfolge für integrierte Software-Komponente

Erfolgt die Abarbeitung des Prozesses ohne Unterbrechung, ist die Prozesslaufzeit die Zeitspanne zwischen Start und Ende der Ausführung des Prozesses. Kommt es zu einer Unterberechnung der Abarbeitung, ist die Summe der Abarbeitungszeiten zwischen Start und Ende der Ausführung des Prozesses die Prozesslaufzeit.

Motorsteuerung

Die maximale Laufzeit eines Prozesses darf bei kooperativen Tasks 200 µs betragen und bei preemptiven Tasks 100 µs, eine Übersicht enthält die Tabelle 2.1. Für Software-Funktionen mit längerer Laufzeit müssen diese auf mehrere Prozesse aufgeteilt werden. Beachtet werden muss, dass ein preemptiver Task jederzeit die langsameren kooperativen Tasks unterbrechen kann.

Tabelle 2.1: maximale Prozesslaufzeiten

Nr.	Prozess-Container	Rechenraster	preemptive/ kooperative Task	maximale Laufzeit
1	void _INI(void)	Ini	-	200 µs
2	void _sync(void)	Syncro	preemptive	10 µs
3	void _1ms(void)	1 ms	preemptive	10 µs
4	void _10ms(void)	10 ms	preemptive	100 µs
5	void _20ms(void)	20 ms	kooperative	200 µs
6	void _100ms(void)	100 ms	kooperative	200 µs
7	void _1000ms(void)	1000 ms	kooperative	200 µs

2.3 Qualitätssicherung von Steuergeräte-Software

Die Sicherung der Qualität von eingebetteten Systemen, gerade im Automobilbau, spielt eine immer größere Rolle. Dies resultiert vor allem aus den Erfahrungen der letzten Jahre, in denen fehlerhafte Software zu kostspieligen Rückrufaktionen geführt hat [44]. Daraus folgten zunehmend Maßnahmen, um eine höhere Softwarequalität zu erzielen, das heißt als Ziel formuliert, eine fehlerfreie Software zu entwickeln. Dieses Ziel bleibt jedoch auch auf längere Zeit für so komplexe Systeme wie eine Motorsteuerung aufgrund der Vielzahl von Fehlerquellen unerreichbar. Aus diesem Grund wird es auch in Zukunft notwendig sein, einen Teil der Entwicklungsleistung in das Auffinden von Fehlern zu investieren, um das Ziel einer *weitgehend* fehlerfreien Software zu erfüllen.

Die Qualitätssicherung umfasst dabei alle Aktivitäten zur Überprüfung von Prozessen und Produkten im Bezug auf Qualitätsziele oder Qualitätsanforderungen. Das Testen als eigenständige Tätigkeit innerhalb des Entwicklungsprozesses ist dabei ein akzeptierter Bestandteil der Qualitätssicherung geworden. Das Ergebnis eines Tests [13] ist, wenn eine hohe Qualität als Erfüllung der Anforderung gesehen wird, die vorhandene Qualität aufzuzeigen und eine Verbesserung zu ermöglichen. Nach der DIN 66272 (identisch mit ISO/IEC 9126) wird der Begriff Softwarequalität wie folgt definiert:

Definition 2.7 Softwarequalität [DIN 66272] [30]**:** Softwarequalität ist die Gesamtheit der Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Dabei wird nach DIN 66272 zwischen folgenden Qualitätsmerkmalen unterschieden: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit.

Für die Beurteilung der Gesamtqualität des Softwareproduktes müssen alle Qualitätsmerkmale beim Testen berücksichtigt werden. Hierfür müssen geeignete Tests zur Überprüfung der Anforderungen gestellt werden.

In der Fachliteratur ist oftmals die folgende Definition des Korrektheitsbegriffes nach Liggesmeyer zu finden [28]:

- Korrektheit besitzt keinen graduellen Charakter, d. h. eine Software ist entweder korrekt oder nicht korrekt.
- Eine fehlerfreie Software ist korrekt.
- Eine Software ist korrekt, wenn sie konsistent zu ihrer Spezifikation ist.
- Existiert zu einer Software keine Spezifikation, ist keine Überprüfung der Korrektheit möglich.

Die Nichterfüllung einer festgelegten Forderung wird nach DIN 66271 [29] als Fehler definiert, die Norm lässt darüber hinaus eine Differenzierung zwischen Fehler und Abweichung zu. Eine Abweichung ist der Unterschied zwischen einem Merkmalswert oder dem Merkmal zugeordneten Wert und einem Bezugswert. Der geforderte Wert kann ein bestimmtes Verhalten der Software sein, die Abweichung damit ein vom geforderten abweichendes Verhalten. Daraus folgt für den Auftraggeber, die Anforderungen für die Qualitätsmerkmale genau festzulegen.

Dem Begriff Fehler kommen (nach DIN 66271 [29]) unterschiedliche Bedeutungen zu, wie Fehlhandlung, Fehlzustand und Fehlfunktion. Nach Scholz [24] wird zwischen den Begriffen „Fehlverhalten“, „Fehler“ und „Irrtum“ unterschieden. In englischsprachigen Publikationen existieren drei unterschiedliche Ausdrücke hierfür:

- Failure: Es handelt sich um ein Fehlverhalten eines Programms, das während seiner Ausführung tatsächlich auftritt (Fehlverhalten, Fehlerwirkung, äußerer Fehler).
- Fault: Es handelt sich um eine fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann (Fehler, Fehlerzustand, innerer Fehler).
- Error: Es handelt sich um eine fehlerhafte Aktion, die zu einer fehlerhaften Programmstelle führt (Irrtum, Fehlhandlung).

Weiterhin lassen sich auf Basis dieser Unterscheidung folgende Feststellungen ableiten: Fehler (errors) bei der Programmentwicklung können zu Fehlern (faults) in einem Programm führen, die ihrerseits Fehler (failure) bei der Programmausführung bewirken können. Die konstruktive Qualitätssicherung reduziert menschliche Fehler (errors). Die analytische Qualitätssicherung entdeckt Programmfehler (faults). Testen löst Laufzeitfehler aus (failures) und führt zur Entdeckung von Programmfehlern (faults).

2.3.1 Maßnahmen zur Software-Qualitätssicherung

In der Arbeit von Fleisch [32] werden die Maßnahmen zur Qualitätssicherung in konstruktive und analytische Qualitätssicherung aufgeteilt (s. Tabelle 2.2). Alle Maßnahmen und Hilfsmittel, die im Laufe der Software-Entwicklungsphasen eingesetzt werden, um Fehler zu vermeiden, werden zur konstruktiven Qualitätssicherung gezählt.

Tabelle 2.2: Beispiele für Qualitätssicherungsmaßnahmen [32]

	Konstruktive Maßnahmen	Analytische Maßnahmen
Entwicklungsprozess-bezogen	- Vorgehensmodelle (z. B. V-Modell) - Standards und Normen (z. B. ISO 9000, CMM)	- Audits - Metriken - Zertifizierung
Produkt-bezogen	- Entwicklungsmethoden (z. B. OOA, OOD) - Standards und Normen (z. B. IEC61508) - CASE-Werkzeuge (z. B. ASCET-SD, Rose)	- Reviews - Testen - Simulation - Formale Verifikation - Rapid Prototyping

Alle Maßnahmen und Hilfsmittel, die zum Entdecken und Beheben von Fehlern in bereits erstellten (Teil-) Produkten der Software-Entwicklungsphasen eingesetzt werden, gehören zur analytischen Qualitätssicherung. Weiterhin werden diese Maßnahmen in Entwicklungsprozess-bezogene und Produkt-bezogene Maßnahmen, entsprechend der Tabelle 2.2, unterschieden. Für die Qualitätssicherungsmaßnahmen werden dabei Normen oder Werkzeuge angegeben. Die Maßnahmen und Hilfsmittel zum Aufdecken von Fehlern durch analytische Qualitätssicherungsmaßnahmen für Software-Komponenten werden im nächsten Abschnitt eingeführt.

2.3.2 Analytische Maßnahmen

Techniken, die in der Lage sind ein Programm bezüglich der Korrektheit zu überprüfen, werden allgemein unter dem Begriff Prüftechnik geführt. Prüfung soll wie folgt definiert werden:

Definition 2.8 Prüfung [DIN 1319-1] [36]**:** Feststellen, inwieweit ein Prüfobjekt eine Forderung erfüllt. Mit dem Prüfen ist immer der Vergleich mit einer Forderung verbunden, die festgelegt oder vereinbart sein kann.

Nach Liggesmeyer [28] wird folgendes Klassifikationsschema (s. Tabelle 2.3) für Software-Prüftechniken vorgeschlagen, um die Korrektheit eines Programms zu überprüfen. Das vorgeschlagene Klassifikationsschema gliedert sich in zwei Klassen, die statischen und die dynamischen Prüftechniken. Wesentliches Unterscheidungsmerkmal ist, ob das zu überprüfende Programm zur Ausführung gebracht werden muss (dynamisch) oder nicht (statisch). Im Wesentlichen sind die dynamischen Prüftechniken in der industriellen Praxis vertreten, einen hohen Stellenwert haben hierbei die funktionsorientierten und die kontrollflussorientierten Prüftechniken.

Für Echtzeitsysteme muss neben der Funktionalität auch das temporale Verhalten der Software geprüft werden. Eine Differenzierung zwischen Funktionalität und Zeitverhalten erfolgt in funktionalen und nicht funktionalen Eigenschaften. Die Performance bzw. die Rechenzeit wird meist den nichtfunktionalen Eigenschaften zu gerechnet [13].

Tabelle 2.3: Prüftechniken [28]

statische Prüftechniken	dynamische Prüftechniken	
Verifizierend	strukturorientiert	funktionsorientiert
- formal	- Anweisungsüberdeckung	- funktionale ÄKB[a]
- symbolisch	- Zweigüberdeckung	- zustandsbasierte Test
Analysierend	- Bedingungsüberdeckung	diversifizierend
- Maße	- strukturierter Pfadtest	- Regressionstest
- Grafiken	- Pfadüberdeckung	Bereichstest
- Review	- Boundary Interior Pfadtest	- Grenzwertanalyse
- Inspektionen	datenflussorientiert	
- Datenflussanomalieanalyse	- Datenkontextüberdeckung	

a - Äquivalenzklassenbildung (s. Abschn. 2.4.1)

2.4 Testen von Steuergeräte-Software

In Anlehnung an Spillner und Linz [13] wird hier unter Testen die Aufgabe des gezielten und systematischen Aufdeckens von Fehlerwirkungen, die auf Defekte hinweisen, verstanden. Testen von Software ist die kontrollierte (im Allgemeinen stichprobenartige) Ausführung eines Testobjektes, die der Überprüfung des Testobjektes dient. Der Test erfolgt unter definierten Bedingungen, bei dem das beobachtete und aufgezeichnete Ist-Verhalten mit dem Sollverhalten (dem spezifizierten Verhalten) des Testobjektes verglichen wird. Die abschließende Bewertung dient der Prüfung, ob das Testobjekt die nach der Spezifikation geforderten Eigenschaften erfüllt, d. h. die Bewertung ob eine korrekte Umsetzung der Anforderungen erfolgte.

2.4.1 Funktionsorientiertes Testen

Die Prüfung der Funktionalität ist die Aufgabe des funktionsorientierten Tests. Die Vollständigkeit des Tests wird durch die Abdeckung der Spezifikation bestimmt. Zu den Problemen des funktionsorientierten Tests gehören das Risiko von ungetestetem Programmcode und der hohe Aufwand für die Testfallerstellung. Es existieren verschiedene Ansätze zur Testauswahl, wie:

Funktionale Äquivalenzklassenbildung (ÄKB**):** Bei der funktionalen ÄKB wird der Eingabewertebereich in Äquivalenzklassen unterteilt. Dabei sind Äquivalenzklassen gleichartige Betriebssituationen mit gleichartiger Wirkung. Aus einer Äquivalenzklasse werden geeignete repräsentative Stellvertreter als Testfälle zur Verfügung gestellt. Für jeden Eingabeparameter müssen wenigstens eine gültige und eine ungültige Äquivalenzklasse gebildet werden. Die Auswahl von Testfällen erfolgt nach gültigen Eingaben und ungültigen Eingaben für die Äquivalenzklasse.

Grenzwertanalyse: Bei der Grenzwertanalyse erfolgt der Test bzw. Überprüfung der Grenzwerte, da für diese Eingaben eine hohe Fehlerhäufigkeit erwartet wird. Die Testfälle werden aus den Grenzen der einzelnen Äquivalenzklassen gebildet.

Zustandsbezogener Test: Zustandsbezogene Tests werden zum Testen gedächtnisbehafteter Software verwendet. Getestet werden alle Zustände sowie alle Zustandsübergänge. Die Basis bieten Zustandsübergangstabellen, diese sind auch zur Testfalldefinition von Fehlsituationen geeignet.

2.4.2 Strukturorientiertes Testen

Der strukturorientierte Test basiert auf der Kontrollstruktur bzw. auf dem Kontrollfluss der Software. Die Testgrundlage ist der sogenannte Kontrollflussgraph, der aus der statischen Codeanalyse hervorgeht. Anhand des Kontrollflusses erfolgen die Kontrolle der Testdaten und die Definition weiterer Testdaten.

Statische Codeanalyse: Die Verwendung graphischer und tabellarischer Darstellungen von Software-Strukturen unterscheidet sich nach Einsatzgebiet zum Beispiel in:

- Kontrollflussgraphen für strukturorientierte, dynamische Testtechniken im Modultest und
- Aufrufgraphen zur Darstellung der Software-Architektur im Integrationstest.

Graphen bestehen aus einer Menge von Objekten (im folgenden Knoten genannt) und Relationen zwischen diesen Objekten (im folgenden Kanten genannt) [19].

Definition 2.9 Kontrollflussgraph: Der Kontrollflussgraph ist ein gerichteter Graph $G = (N, E, n_{start}, n_{stop})$. N ist die endliche Menge der Knoten und E die zweielementige Teilmenge von $N \times N$. $E \subseteq N \times N$ ist die Menge der gerichteten Kanten. $n_{start} \in N$ ist der Startknoten. $n_{stop} \in N$ ist der Endeknoten.

Die Knoten stellen Anweisungen dar. Eine gerichtete Kante von einem Knoten *i* zu einem Knoten *j* beschreibt einen möglichen Kontrollfluss von Knoten *i* zu Knoten *j*. Alternativ werden gerichtete Kanten auch als Zweige bezeichnet. Eine Sequenz aus Knoten $(n_{start}, n_1, ..., n_m, n_{stop})$, die mit dem Startknoten n_{start} beginnt und mit dem Endeknoten n_{stop} endet, heißt Pfad p. Für das in Abb. 2.6 dargestellte Programm besteht ein möglicher Programmpfad aus den Knoten $n_{start}, n_1, n_2, n_{stop}$. Für das Beispielprogramm können unter der Eingabe von Testdaten drei mögliche Pfade erreicht werden (s. Tabelle 2.4).

```
if (r)                //n1
  status = false;     //n2
else
  if (s)              //n3
    status = true;    //n4
```

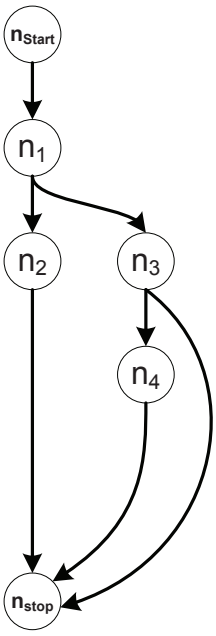


Abb. 2.6: Programm mit zugehörigen Kontrollflussgraphen

Es ist zu beachten, dass ein Kontrollflussgraph keine Datenabhängigkeiten von Berechnungen darstellen kann. Das heißt, dass nur eine Darstellung der möglichen Programmpfade möglich ist. Weiterhin sind zwei Pfade nur identisch, falls die Sequenz ihrer Knoten identisch ist.

Tabelle 2.4: Aufzählung der möglichen Programmpfade

Pfad-Nr.	Eingabe {r,s}	Programmpfad	Ergebnis {status}
1	1,0;1,1	$n_{start}, n_1, n_2, n_{stop}$	false
2	0,1	$n_{start}, n_1, n_3, n_4, n_{stop}$	true
3	0,0	$n_{start}, n_1, n_3, n_{stop}$	{status} unverändert

Vereinfachung des Kontrollflussgraphen: Solange der Kontrollfluss erhalten bleibt, können Teile des Programms zu Segmenten zusammengefasst werden. Ein Segment (Grundblock, Basisblock) ist eine Folge von Knoten und Kanten mit maximaler Länge und mit den Eigenschaften:

- Das Segment kann ausschließlich über den ersten Knoten des Segments betreten werden.
- Wird der erste Knoten des Segments durchlaufen, werden alle Knoten des Segments in der vorgesehenen Reihenfolge genau einmal durchlaufen.

Segmente sind somit Anweisungssequenzen ohne Verzweigung. Ein modifizierter Kontrollflussgraph (Basisblockgraph) fasst Anweisungen eines Segmentes zu einem Knoten zusammen. Die Visualisierung der kompletten Struktur einer Software-Komponente kann als Funktionsaufrufgraph aus Knoten, die die Details der Funktionen verstecken, dargestellt werden. Werden diese Funktionen entfaltet, öffnet sich der Basisblockgraph der entsprechenden Funktion. Ein Basisblock selbst ist ein gefalteter Knoten, welcher nach Entfaltung die Anweisungsliste des Basisblocks anzeigt. Durch Falten, Ein- und Ausblenden von Informationen werden logische Sichten auf die Ebenen der Software-Struktur erzeugt. Diese Darstellung hilft bei Programmen mit sehr vielen Programmpfaden.

Testüberdeckung: Die Vollständigkeit des Tests wird durch die erreichte Abdeckung des Quellcodes beurteilt. Hierbei wird zum Beispiel unterschieden**:**

- Die Anweisungsüberdeckung (engl. statement coverage test) erfordert die mindestens einmalige Ausführung aller Anweisungen der zu testenden Software. Das C0-Maß dient zur Berechnung der Überdeckung der Anweisungen.
- Die Zweigüberdeckung erfordert die Ausführung aller Zweige der zu testenden Software. Das C1-Maß (s. Gl. 2.8) dient zur Berechnung der Überdeckung der Zweige.

$$C1 = \frac{\text{Anzahl durchlaufene Zweige}}{\text{Gesamtzahl Zweige}} * 100\% \tag{2.8}$$

- Der Pfadüberdeckungstest (engl. path coverage) erfordert die Ausführung aller unterschiedlichen vollständigen Pfade der zu testenden Software (C2-Maß).

2.4.3 Messen der Prozesslaufzeit

Eine weitere mit dem Test durchzuführende Tätigkeit ist das Messen. Nach DIN 1319-1 wird der Begriff Messung wie folgt erklärt:

Definition 2.10 Messung [DIN 1319-1] [36]**:** Messung einer Messgröße ist die Ausführung von geplanten Tätigkeiten zum qualitativen Vergleich der Messgröße mit einer Einheit.

Die Tätigkeiten zum Erfassen der Messwerte sind im Allgemeinen experimenteller Natur, können jedoch theoretische Berechnungen mit einschließen. Unter einem Messwert wird der Wert, der zur Messgröße gehört und der Ausgabe eines Messgerätes zugeordnet ist, verstanden. Die Auswertung von Messwerten bis zum Ergebnis wird zur Messung gezählt, nicht jedoch die Feststellung, ob das Ergebnis einer Forderung genügt, dies gehört zum Prüfen. Die Prozesslaufzeit t kann aus der Messung der benötigten Taktzyklen c für die Ausführung eines Programms nach Gl. (2.9) berechnet werden.

$$t = c * t_{Zyklen} \tag{2.9}$$

Die Prozesslaufzeit t ist damit das Ergebnis aus einem Messwert, eine Prüfung findet bis hierher nicht statt. Die Prüfung erfolgt anschließend auf der Basis der Ergebnisse aus der Messung. Die Messmethode bestimmt die Art des Vorgehens bei der Messung, für die Prozesslaufzeit t muss zum Beispiel geklärt werden, ob die Messgröße direkt oder indirekt erfasst werden kann. Zu den fundamentalen Messprinzipien [43] gehören das direkte und das

indirekte Messen. Die direkte Messung setzt die Zugänglichkeit des Messobjektes voraus. Das Messobjekt ist der Träger der Messgröße. Im vorliegenden Fall ist das Programm, bzw. abstrakter formuliert die Software-Komponente mit der darunter liegenden Hardware, das Messobjekt.

Messabweichung: Die Zusammenhänge zwischen dem am Messobjekt gemessen Wert und dem wahren Wert werden in der Abb. 2.7 dargestellt. Um den Erwartungswert μ gruppieren sich in Form einer Häufigkeitsverteilung die unter Wiederholbedingungen aufgenommenen Messwerte einer Messgröße, hier als normale Häufigkeitsdichtefunktion[4] skizziert. Je größer die Anzahl der Messwerte ist, umso weniger wahrscheinlich ist die Abweichung des arithmetischen Mittelwerts vom Erwartungswert. Der Erwartungswert weicht vom unbekannten wahren Wert x_W um die systematische Messabweichung e_s ab.

In Abb. 2.7 eingezeichnet ist ein einzelner Messwert x. Er verfehlt den wahren Wert um die Messabweichung, die sich additiv aus systematischer Messabweichung e_s und zufälliger Messabweichung e_r zusammensetzt. Der Messwert x ergibt sich damit aus:

$$x = x_w + e_r + e_s \tag{2.10}$$

Die systematische Messabweichung e_s setzt sich aus einem bekannten ($e_{s,b}$) und einem unbekannt bleibenden Anteil ($e_{s,u}$) zusammen.

$$e_s = e_{s,b} + e_{s,u} \tag{2.11}$$

Zur bekannten systematischen Messabweichung $e_{s,b}$ trägt z. B. auch die bei einer früheren Kalibrierung des benutzten Messgerätes festgestellten systematischen Messabweichung des Messgerätes bei.

Der bekannte Anteil der systematischen Messabweichung kann mit umgekehrtem Vorzeichen als Korrektion[5] zum Messwert x addiert werden. Damit findet man den berichtigten Messwert x_E. Er weicht vom wahren Wert nur noch um die Summe aus dem unbekannt bleibenden Anteil $e_{s,u}$ der systematischen Messabweichung und der nicht genau feststellbaren zufälligen Messabweichung e_r ab.

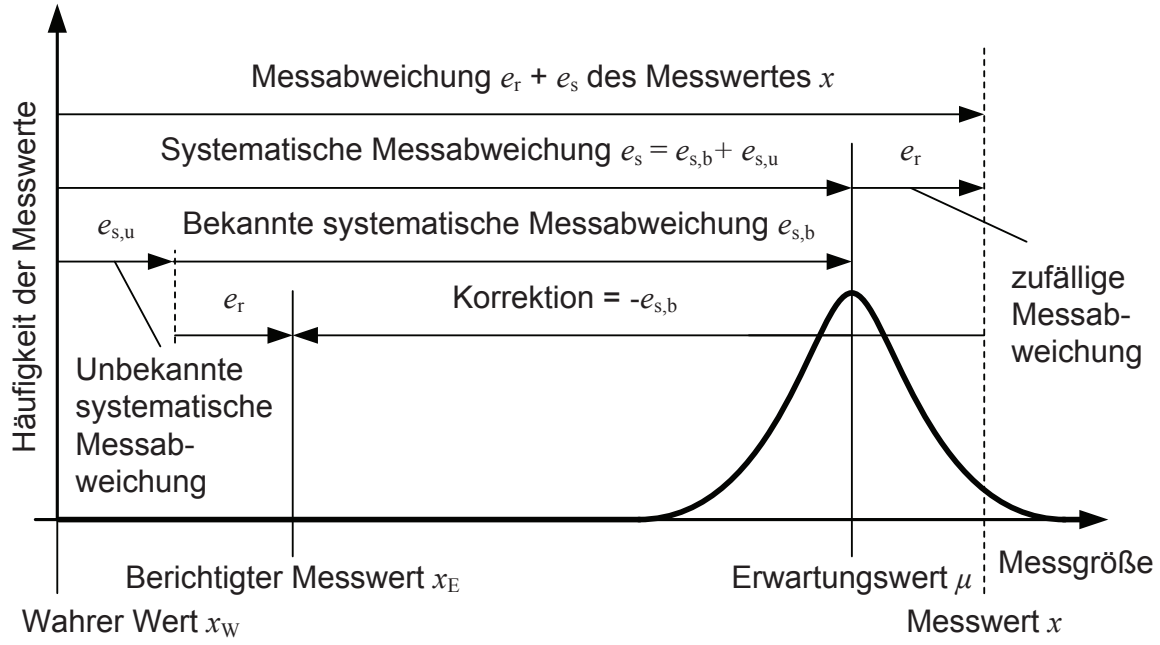


Abb. 2.7: Häufigkeitsverteilung unter Wiederholungsbedingungen [36]

[4] Die Voraussetzungen für eine annähernd normalverteilte Zufallsgröße (Messgröße) werden in [20] beschrieben.
[5] Der Begriff „Korrektion“ wird hier nach DIN 1319-1 [36] verwendet.

2.5 Prüfstrategie für Motorsteuergeräte-Software

In dieser Arbeit wird die temporale Prüfung von Software-Funktionen als Bestandteil der Qualitätssicherung betrachtet. Daher sollen zunächst die Elemente der analytischen Qualitätssicherung vorgestellt werden sowie der organisatorische und technische Rahmen, in dem sich die temporale Prüfung einbetten muss. Hierbei werden auch etablierte unternehmensübergreifende Prozesse betrachtet.

Die Software-Prüfung für eingebettete Echtzeitsysteme lässt sich grob in informale Techniken und formale Techniken untergliedern. Informale Techniken, wie Reviews, Inspektionen und dynamische Testtechniken, finden in der Automobilindustrie eine hohe Verbreitung. Der große Vorteil der informalen Qualitätssicherungstechnik liegt in der Einfachheit und universellen Anwendbarkeit, selbst komplexe Software-Systeme können durch systematische, dynamische Tests geprüft werden. Der große Nachteil beim dynamischen Test liegt im Stichprobencharakter und damit in der Unvollständigkeit der Ergebnisse. Die systematischen, dynamischen Tests sind meist nicht erschöpfend (vollständig) möglich, d. h., Restfehler können nicht ausgeschlossen werden. Zwischen Software mit geringen und hohen Sicherheits- und Zuverlässigkeitsanforderungen wird daher beim Test stark differenziert. So werden unterschiedlich hohe Restrisiken je nach Art der Software akzeptiert.

Um die Akzeptanz der temporalen Prüfung zu erreichen, soll der Aufwand der bisherigen Prüfstrategie nicht wesentlich erhöht werden, daher sollen Synergien zu der bestehenden Software-Prüfung genutzt werden. So fallen in jeder Testphase Testdaten an, deren Eignung für die temporale Prüfung der Software-Funktion, im Rahmen dieser Arbeit bewertet werden soll. Hierzu gehören Software-Strukturinformationen aus der statischen Codeanalyse genauso wie die bereits anfallenden Testfälle aus den funktions- und strukturorientierten Tests. Nachfolgend wird die Ausgangssituation zur Absicherung von Steuergeräte-Software dargestellt und die bestehenden Unzulänglichkeiten vorgestellt, im Speziellen für die Motorsteuergeräte-Software.

2.5.1 Testverfahren

Bei der modellbasierten Funktionsentwicklung wird grundsätzlich zwischen drei entwicklungsbegleitenden Prüfverfahren unterschieden: der Simulation (Offline-Test), dem Rapid Prototyping (Online-Tests) sowie dem Test im Fahrzeug mit Mustersteuergeräten. Der Offline-Test oder auch das modellbasierte Testen prüft das mathematische Modell der Fahrzeugfunktion durch entsprechende Testgeneratoren, welche Eingangsparameter manipulieren und dadurch bestimmte Software-Zustände stimulieren. Die Ausgangsdaten der Zustandsautomaten und Regelungsfunktionen werden dann gegen ihre Sollwerte verglichen. Beim Online-Test werden die zu testenden Module in ein Rapid Prototyping System integriert, mit dem Steuergerät verbunden und zusammen am HiL-Simulator oder im Fahrzeug auf seine Reaktionen getestet. Die Ein- und Ausgangsdaten werden dabei mit einem Mess- und Applikationswerkzeug[6] aufgezeichnet. Der Test im Fahrzeug dient dem Prüfen der Software gegen reale Einsatzbedingungen. Die Fahrzeugfunktion wird hier nach ihrer Integration in die Steuergeräte-Software auf dem Mustersteuergerät im Steuergeräteverbund geprüft. Im kompletten Testablauf werden alle drei Testverfahren auf das Modul angewandt.

Der Offline- und Online-Test eignen sich nicht für eine temporale Prüfung, wenn der erzeugte Programmcode sich von dem Code für das Zielsystem unterscheidet. Dies ist häufig der Fall, da die eingesetzten Codegeneratoren spezielle Optimierungen für das Simulations- oder Prototypensystem erzeugen.

[6] Mess- und Applikationswerkzeuge können sämtliche Messgrößen zeitlich zueinander korreliert darstellen und aufzeichnen.

Der Test auf Mustersteuergeräten hat für die temporale Prüfung eine hohe Relevanz, da der Programmcode alle Optimierungen für das Zielsystem aufweist und die Hardware sich nur geringfügig vom Zielsystem unterscheidet (dies gilt zumindest für Prozessor- und Speicherarchitektur).

2.5.2 Testphasen

Der Test einer Fahrzeugfunktion als eingebettete Software wird, wie auch der Test herkömmlicher Software, in verschiedene Testphasen unterteilt. Dabei wird im Allgemeinen zwischen dem Code-, dem Modul-, dem Integrations- und dem Abnahmetest unterschieden. Abbildung 2.8 zeigt den Software-Entwicklungsprozess, den eine Software-Funktion von der Anforderungsspezifikation bis hin zum Abnahmetest durchläuft. Parallel zu den Entwicklungsphasen verlaufen gemäß dem V-Model die Testphasen zur Qualitätssicherung. Die Methoden und Maßnahmen sind dabei an der IEC 61508-3 Anhang A und B [81] orientiert.

Nach dem Software-Abnahmetest folgen der bereits im Kapitel 1 eingeführte Steuergeräte-Integrationstest und der Steuergeräte-Abnahmetest (s. Abb. 1.1), in der Einführung allgemein als Steuergerätetest und Abnahmetest bezeichnet. Diese Testphasen beziehen sich nicht explizit auf die Motorsteuergeräte-Software, vielmehr geht es hier um das Motorsteuergerät als Gesamtsystem.

Ausgangspunkt für die produktbezogenen Maßnahmen zur Qualitätssicherung ist das der Fahrzeugfunktion zugrunde liegende Modell bzw. der daraus generierte Programmcode. Das Entwerfen der Fahrzeugfunktionen mit CASE Tools wie ASCET [26] ermöglicht schon während der Modellierung eine genaue Struktur-, Verhaltens- und Datenbeschreibung.

Software-Modulentwurf: Die Absicherung der Software erfolgt bereits im Software-Modulentwurf auf Basis diverser Maßnahmen wie Simulation, Prototypenerstellung, Inspektion des Software-Modulentwurfs gegen Modellierungsrichtlinien (wie die Beschränkung des Sprachumfanges). Des Weiteren erfolgt eine Prüfung auf das Einhalten von Namenskonventionen oder etwa auf die Berücksichtigung der bekannten Einschränkungen (wie die Begrenzung von Schleifeniterationen) und Fehler des Modellierungswerkzeugs [28].

Software-Codetest: Der Programmcode muss trotz automatischer Generierung getestet werden, da meist keine Zertifizierung des Codegenerators erfolgt. Das heißt, der Codegenerator-Hersteller übernimmt keine Garantie für die Korrektheit der Abbildung vom Funktionsmodell auf den generierten Programmcode. Der Programmcode wird mittels statischer Codeanalyse auf Einhaltung der Codierungsrichtlinien, Berücksichtigung der bekannten Einschränkungen und Fehler des Compilers und der Software-Codemetriken [82] überprüft. Die zulässigen Werte der Software-Codemetriken werden in Codierungsrichtlinien festgelegt, welche Bestandteil der Anforderungsspezifikation sind.

Software-Modultest: Beim Software-Modultest wird zwischen Black- und White-Box-Test unterschieden. Der Black-Box-Test überprüft die Funktionalität des Programms, aber nicht dessen Arbeitsweise. Es wird dabei nur über vorgegebene Schnittstellen mit dem Programm bzw. dem System kommuniziert. Wichtige Bestandteile des White-Box-Tests sind die Abdeckungstests und das Durchführen von Testfällen aus der Fehlererwartung. Die Abdeckungsgrade müssen dabei auf Quellcode-Ebene ermittelt werden.

Der Black-Box-Test besteht aus den Tests gegen sämtliche Anforderungen und den externen Schnittstellentests, welche zum Beispiel aus einer Grenzwertanalyse [13] abgeleitet werden. Ein expliziter Leistungstest, der die Laufzeit ermittelt und prüft, erfolgt nicht. Diese Testphase scheint jedoch geeignet zur Ermittlung von Laufzeiten, da hier eine hohe Test- bzw. Codeabdeckung gefordert wird.

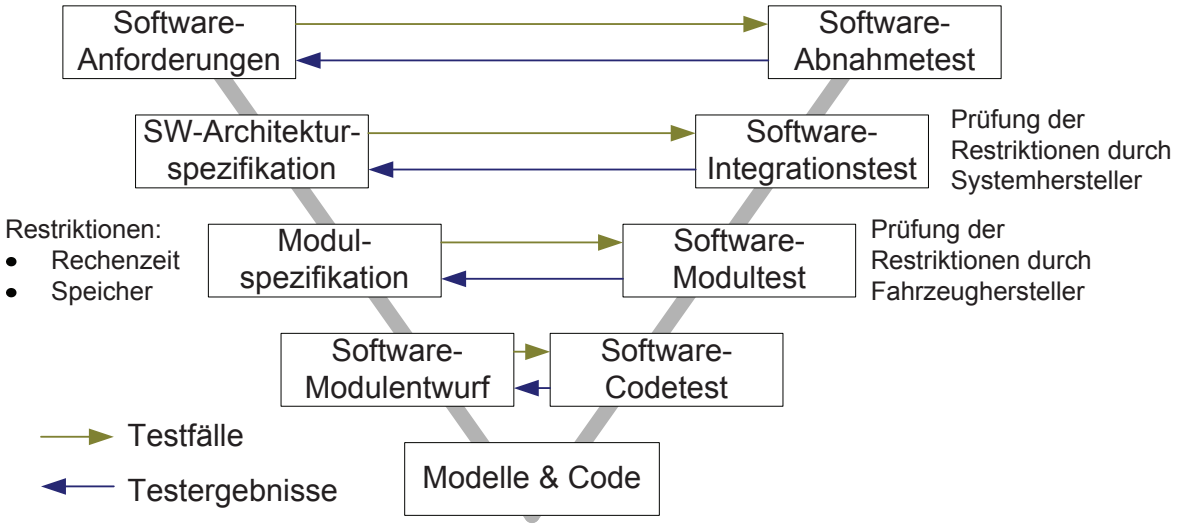


Abb. 2.8: Software-Entwicklungsprozess beim Fahrzeughersteller

Software-Integrationstest: Der Software-Integrationstest besteht aus White- und Black-Box-Test in Hinsicht auf externe und interne Schnittstellen und dem Leistungstest. Beim Software-Integrationstest wird die gesamte Software getestet, um so das Zusammenwirken der einzeln getesteten Module zu verifizieren. Es müssen der Normal- und Maximalbedarf an Prozessorleistung und daraus resultierende Reaktionszeit, Busbandlasten und Speicherbedarf (RAM, ROM für Programm und Daten) geprüft werden. Es muss durch den Systemlieferanten sichergestellt werden, dass die Hardware-Architektur diesen Anforderungen mit ausreichender Toleranz gerecht wird. Der Ressourcentest erfolgt darüber hinaus zur Absicherung der Einhaltung von zugewiesenen Ressourcen von fremdentwickelter Software. Die Ressourcenverfolgung liegt beim Systemlieferanten, dem Systemintegrator. Hier werden erstmals temporale Aspekte explizit berücksichtigt.

Software-Integrationstest (White-Box-Test) beim Fahrzeughersteller - Testintegration: Die Testfälle sollten sämtliche Anforderungen an die Software-Komponente abdecken. Für die Testintegration werden vom Systemhersteller spezielle Programmstände zur Verfügung gestellt, die die beschriebenen Schnittstellen für die neuen Komponenten bereithalten. Die internen Schnittstellentests weisen die Konsistenz der internen Schnittstellen zwischen den Software-Modulen nach. Dies geschieht implizit während des Übersetzens und Bindens (Linken) der Software-Module zu Software-Komponenten. Das Linken in einen Teststand wird verhindert, wenn der definierte RAM- und Flash-Bereich erschöpft ist. Dies ermöglicht eine implizite Speicherverfolgung. Die externen Schnittstellentests erfordern Testfälle, die die Konsistenz der Daten, die über die Schnittstellen ausgetauscht werden, nachweisen. Die Testfälle können z. B. aus einer Grenzwertanalyse abgeleitet werden. Dabei müssen sowohl die Grenzwerte der Datentypen als auch die plausiblen Wertebereiche der Schnittstellenvariablen betrachtet werden. Eine repräsentative Auswahl der Testfälle wird an den Systemlieferanten zum Software- und Steuergeräte-Integrationstest übergeben. Der geforderte Abdeckungsgrad verlangt mindestens einen einmaligen Aufruf jeder Software-Funktion (Aufrufabdeckung). Für die Messungen der tatsächlichen Systemauslastung durch die fremdentwickelten Funktionen benötigt der Systemlieferant vom Fahrzeughersteller eine Beschreibung, wie die neuen Funktionen in den Zustand gebracht werden, in dem sie die höchste Laufzeitbelastung erzeugen. Der Fahrzeughersteller muss einen Testfall für den Leistungstest beim Systemhersteller bereitstellen. Hier sollen die in dieser Arbeit entwickelten Verfahren und Methoden helfen, indem sie verlässliche Hinweise auf kritische Pfade und Laufzeiten geben.

Software-Integrationstest (Black-Box-Test) beim Systemlieferanten – Serienintegration: Die Testfälle enthalten eine Auswahl aus den Anforderungen an die jeweilige fremdentwickelte Software-Komponente. Dabei muss sichergestellt werden, dass wesentliche Anforderungen an die Software-Komponente abgedeckt werden. Der interne Schnittstellentest und der externe Schnittstellentest erfordern Testfälle, die die Konsistenz der Schnittstellen nachweisen. Dabei müssen wieder die Grenzwerte der Datentypen als auch die plausiblen Wertebereiche der Schnittstellenvariablen betrachtet werden.

Software-Abnahmetest: Der Software-Abnahmetest erfolgt nach der Integration der Software-Komponenten durch den Systemlieferanten. Beim Software-Abnahmetest wird die gesamte Software gegen ihre Anforderungen gemäß der Spezifikation geprüft. Eine abschließende Ressourcenverfolgung zur Dokumentation von Speicher- und Rechenzeitbedarf erfolgt durch den Systemhersteller auf Basis der gelieferten Testfälle, welche bisher allerdings aus funktionalen Tests ohne explizite und vertiefte Berücksichtigung der temporalen Aspekte hergeleitet werden.

2.5.3 Regressionstest

Jede Testphase gliedert sich in ihre jeweiligen Testaktivitäten. Dadurch wird strukturelles und reproduzierbares Testen ermöglicht. Der systematische Test gliedert sich im Allgemeinen in die Aktivitäten Testfallermittlung, Testdurchführung, Monitoring[7] und Testauswertung. Der Testprozess als Bestandteil des Funktionsentwicklungsprozesses gliedert sich in die Aktivitäten Testplanung, Testspezifikation, Testdurchführung, Testdokumentation und Testauswertung. Die Reproduzierbarkeit von Testergebnissen sowie die Verwahrung der Testdokumente werden aus haftungsrechtlichen Gründen gefordert. Dies gilt sowohl für die vom Fahrzeughersteller entwickelten Softwarebestandteile als auch für die vom Systemlieferanten erstellten Systemkomponenten. Zu diesem Zweck und zur Kostenreduktion werden frühzeitig die Tests zur Wiederverwendung als Regressionstests ausgelegt.

Der Regressionstest ermöglicht eine Wiederholung ausgewählter Testfälle, um etwa Seiteneffekte aus Modifikationen der bereits getesteten Module zu erkennen. Er dient als Nachweis, dass Modifikationen von Software keine unerwünschten Auswirkungen auf die Funktionalität besitzen, da durch Funktionserweiterung und Fehlerkorrektur neue Fehler entstehen können.

Neu erzeugte Ausgaben werden mit den Ausgaben der Vorläuferversion verglichen. Sind keine Unterschiede aufgetreten, war der Regressionstestfall erfolgreich. Werden Unterschiede festgestellt, ist zu prüfen, ob diese gewünscht oder fehlerhaft sind. Ist das Verhalten gewünscht, dann ist das geänderte Verhalten für zukünftige Regressionstests das Soll-Verhalten und wird zum Referenzfall. Bei fehlerhaftem Verhalten muss der Fehler lokalisiert und korrigiert werden. Ob ein wiederholter Testfall korrekt abgearbeitet wurde, wird nicht durch den Vergleich mit der Spezifikation beurteilt, sondern durch Vergleich der neu erzeugten Ausgaben mit den Ausgaben der Vorläuferversion. Regressionstestwerkzeuge zeichnen die Testeingaben und Testergebnisse bei der Testdurchführung auf. Nachdem die Korrektheit der erzeugten Testergebnisse anhand der Spezifikation verifiziert wurde, bezeichnet man die aufgezeichneten Testeingaben und Testergebnisse als Referenztestfälle. Durch Automatisierung wird der Aufwand für manuelle Wiederholungen der Testfälle eingespart. In der Automobilindustrie wird vor allem der HiL-Simulator für den Regressionstest benutzt. Die beabsichtigte Wiederverwendung der im Rahmen der bisherigen Qualitätssicherung anfallenden Testfälle legt es nahe, den in dieser Arbeit vorgeschlagenen temporalen Test als Regressionstest zu konzipieren.

[7] Beim Monitoring werden abhängig von der jeweiligen Testphase die Ist-Werte der beobachteten Testobjekte für die folgende Testauswertung in entsprechender Form aufgezeichnet oder festgehalten.

2.6 Zusammenfassung

Die Qualität von Echtzeit-Systemen kann nicht allein durch konstruktive Maßnahmen sichergestellt werden. Um Fehler in der Implementierung aufzudecken, sind ebenso analytische Maßnahmen notwendig. In allen Testphasen wird ein funktionsorientiertes Vorgehen beim dynamischen Testen verfolgt. Standard ist hier die funktionale Äquivalenzklassenbildung oder das zustandsbasierte Testen. Funktionales Testen bietet allgemein keine Sicherheit, dass die Testfälle eine vollständige Abdeckung des Programmcodes sicherstellen, daher ist eine strukturorientierte Abdeckung notwendig. Sichergestellt werden muss, dass die Testfälle eine vollständige Abdeckung erzielen, ein Minimum an Abdeckung ist die Anweisungsabdeckung. Für Software mit hohen Sicherheits- und Zuverlässigkeitsanforderungen (z. B. Motorsteuerung) wird eine vollständige Zweigabdeckung gefordert (ISO/WD 26262).

Der strukturorientierte Test stellt hierbei die Kriterien zur Verfügung, über die Vollständigkeit der Tests zu urteilen. Testphasen sind dabei so organisiert, dass die strukturorientierten, dynamischen Tests einmalig in Modultests nach der Fertigstellung des Programmcodes erfolgen. Funktionsorientierte, dynamische Tests werden dagegen in allen Testphasen durchgeführt unter jeweils angepassten Bedingungen. Wirtschaftlich sinnvoll sind diese Tests nur mit Werkzeugunterstützung durchzuführen, daher werden durchgängig HiL-Systeme zur Verwaltung und Wiederverwendung von Testfällen [83] verwendet.

Alle Testaktivitäten sind bisher auf funktionale Tests ausgerichtet. Als Zusatzbedingung wird das Laufzeitverhalten nur in den entwicklungsbegleitenden Prüfverfahren beim Rapid Prototyping (Online-Tests) sowie dem Test im Fahrzeug mit Mustersteuergeräten betrachtet. Hier wird ein temporales Fehlverhalten meist nicht sofort sichtbar und muss erst durch aufwendige Fehlersuche bestimmt werden. Ziel dieser Tests ist es jedoch, die dynamischen Eigenschaften des Echtzeit-Systems unter realen Einsatzbedingungen zu prüfen. Aufgrund der Komplexität der Software von Echtzeit-Systemen gestaltet sich der Test dieser Systeme als umfangreiches Verfahren, welches nur durch einen durchgängig strukturierten und dokumentierten Ablauf zu beherrschen und nachzuvollziehen ist. Der Verzicht auf einen Leistungstest, d. h. eine Laufzeitbewertung im Software-Modultest, birgt die Gefahr, dass temporale Implementierungsfehler erst spät beim Integrationstest entdeckt werden. Der vom Entwickler spezifizierte Testfall für die höchste Laufzeitbelastung durch die neue Software-Funktion wird nicht verifiziert. Dieses Vorgehen resultiert daraus, dass keine Methoden und Werkzeuge verfügbar sind, die dem Entwickler bei der Spezifikation und Überprüfung der Testfälle unterstützen. Die Erstellung des Testfalls setzt Expertenwissen über die Codeumsetzung des Funktionsmodells voraus und stellt darüber hinaus nur eine Stichprobe dar, deren Verlässlichkeit nicht sichergestellt ist.

Die Qualität des Tests hängt damit im starken Maße von der Erfahrung einzelner Entwickler ab. Dies kann der wachsenden Komplexität eingebetteter Software-Systeme im Fahrzeug auf Dauer nicht gerecht werden. Die Ergebnisse dieser Arbeit sollen den Entwicklern die Werkzeuge in die Hand geben, um unabhängig von persönlichen Erfahrungen qualitativ hochwertige Testfälle spezifizieren zu können. Des Weiteren ist eine Beurteilung des Testfalls notwendig. Hierfür muss eine sichere obere Laufzeitgrenze für die getestete Software angegeben werden können, anhand der das Restrisiko beurteilt wird. Die Aufgabe der Ressourcenverfolgung muss so früh wie möglich im Entwicklungsablauf erfolgen. Zum einen bedeutet das, dass die Ressourcenverfolgung mit der analytischen Qualitätssicherung im Software-Modultest und bei der Testintegration in der Musterphase erfolgen muss und zum anderen auch bei der Einführung neuer Steuergerätegenerationen (mit neuen Hardwarearchitekturen), um Fehlentwicklungen frühzeitig entgegen wirken zu können (s. [38]).

3 Stand der Technik bei der Laufzeitanalyse

Dieses Kapitel bietet einen Überblick über die Methoden, Verfahren und Werkzeuge zur Laufzeitanalyse aus aktuellen Forschungsprojekten sowie aus der kommerziellen Entwicklung. Es werden die fundamentalen Ansätze und die für heutige Mikrocontroller noch gültigen und anwendbaren Methoden diskutiert. Der bestehende Anspruch liegt daher darin, Methoden auf das Wesentliche reduziert zu beschreiben und Begriffe einzuführen.

Die Bewertung, die vorgenommen wird, basiert auf Faktoren, die vor allem die industrielle Verwendung und die daraus resultierenden Anforderungen an Methoden und Verfahren zur Laufzeitanalyse von Motorsteuergeräte-Software einbeziehen. Es wird sich dabei auf die wesentlichen Faktoren beschränkt, eine umfassende Darstellung erfolgte im Kontext dieser Arbeit in [52]. Hier werden die Verfahren und Werkzeuge (Tools) zur Bestimmung der Worst-Case Execution Time (WCET) von Software-Komponenten beschrieben.

Bei der Laufzeitbestimmung von Echtzeitprogrammen wird aufgrund der Anforderung nach Rechtzeitigkeit des Ergebnisses der wesentliche Schwerpunkt auf die Ermittlung der WCET gelegt. Hierbei werden in den einzelnen Forschungsgruppen auch die Auswirkungen von Beschleunigungsmechanismen moderner Prozessoren betrachtet.

Das Kapitel ist so aufgebaut, dass sich zu erst mit dem methodischen Hintergrund – der Lösungsansätze zum Problem der Laufzeitbestimmung – beschäftigt wird, und anschließend werden die wesentlichen Ansätze vorgestellt, die diese Methoden zu einem Verfahren zusammenfügen. Die meisten WCET-Tools dienen als Nachweis der Wirksamkeit von Verfahren und Ansätzen. Diese Tools sind oftmals als Prototypen implementiert, die eine Teilaufgabe der Laufzeitbestimmung (wie Cacheanalyse) meist sehr genau abbilden können. Die Begrenzung auf einen Teilaspekt der Problemstellung führte bisher selten dazu, dass diese Tools in den kommerziellen Bereich vorgedrungen sind. Kommerzielle Tools dagegen sind meist auf einen größeren Anwendungsbereich ausgelegt, was nicht selten dazu führt, zwischen Wirtschaftlichkeit und Genauigkeit zu differenzieren. Im nachfolgenden Abschnitt sollen daher diese beiden Aspekte berücksichtigt werden.

Eine grundsätzliche Unterscheidung der Ansätze für die Ermittlung der WCET kann in Verfahren, die den Code ausführen und die Laufzeit messen sowie in Verfahren, die den Code ohne Ausführung interpretieren und mittels eines Architekturmodells auf die Ausführungszeit schließen, erfolgen. Analytische Methoden bedingen eine hinreichend genaue Nachbildung des Zielsystems, aus diesem Modell muss dann ein Simulieren des Worst-Case möglich sein. Analytische Methoden werden aufgrund ihres Arbeitsprinzips den statistischen Prüfmethoden zugeordnet. Verfahren, die auf der Ausführung des Codes basieren und die Laufzeiten messen, werden unter dynamische Prüfmethoden zusammengefasst.

3.1 Dynamische Prüfmethoden zur Laufzeitbestimmung

Zur Laufzeitbestimmung durch Messung werden vor allem klassische Werkzeuge und Verfahren der Software-Entwicklung und des Software-Tests [44] wie Compiler, Hardware- und Software-Debugger, Simulatoren und Emulatoren angewendet. Weitere Möglichkeiten bieten

einige spezielle Betriebssysteme, die Funktionen zur Laufzeitbestimmung anbieten. Die Verfahren können zielsystembezogen wie folgt eingeteilt werden:

- Softwaretechnische Messverfahren wie Betriebssysteme, Hardwaresimulator und Compiler (hier wird die Messkette im Wesentlichen durch softwaretechnische Messeinrichtungen realisiert) und
- Hardwaretechnische Messverfahren wie Hardware-Debugger und Emulatoren und zielsystemeigene Analysefunktionen (hier wird die Messkette aus hardwaretechnischen Messgeräten gebildet).

Ein weiteres Unterscheidungsmerkmal zwischen den Verfahren betrifft das Messobjekt, hier kann unterschieden werden zwischen instrumentierenden Verfahren [54], die eine Modifikation des Messobjektes vornehmen, und nicht instrumentierenden Verfahren, die das Messobjekt unverändert lassen. Die Instrumentierung geschieht meist automatisch durch z.B. einen Compiler, Präprozessor oder manuell durch einen Editor im Quelltext.

In [53] wird eine weitere Unterteilung in „light weight Software Monitoring“ und „heavy weight Software Monitoring“ vorgenommen. Im ersten Fall wird nur eine Instrumentierung zur Messung eingefügt, im zweiten Fall wird zusätzlich eine Manipulation der Beschleunigungsmechanismen des Prozessors vorgenommen. Dies soll bei der Messung einen Worst-Case für die Hardwarearchitekturmerkmale wie Cache und Pipeline erzwingen.

Durch die Modifikation der Produktsoftware wird hieraus eine Messsoftware, die sich in spezifischen Eigenschaften unterscheiden kann. Daher sollen zunächst die Möglichkeiten der Instrumentierung diskutiert werden. Der klassische Ablauf der Messung ist in Abb. 3.1 dargestellt, hier werden die Eingriffsmöglichkeiten wie das Modifizieren des Quellprogramms und die Auswahl der Ausführungsumgebung grau hinterlegt dargestellt. Die Programmlaufzeitanalyse setzt sich aus drei Schritten zusammen:

- Der erste Schritt ist das Instrumentieren des Programms, dies wird von einem Präprozessor übernommen. Anschließend erfolgt das Übersetzen und Binden des Programms.
- Der zweite Schritt ist das Ausführen des Programms auf einem Laufzeitsystem zur Erzeugung von Laufzeitmessdaten.
- Der dritte Schritt ist die Analyse der Daten durch einen Postprozessor.

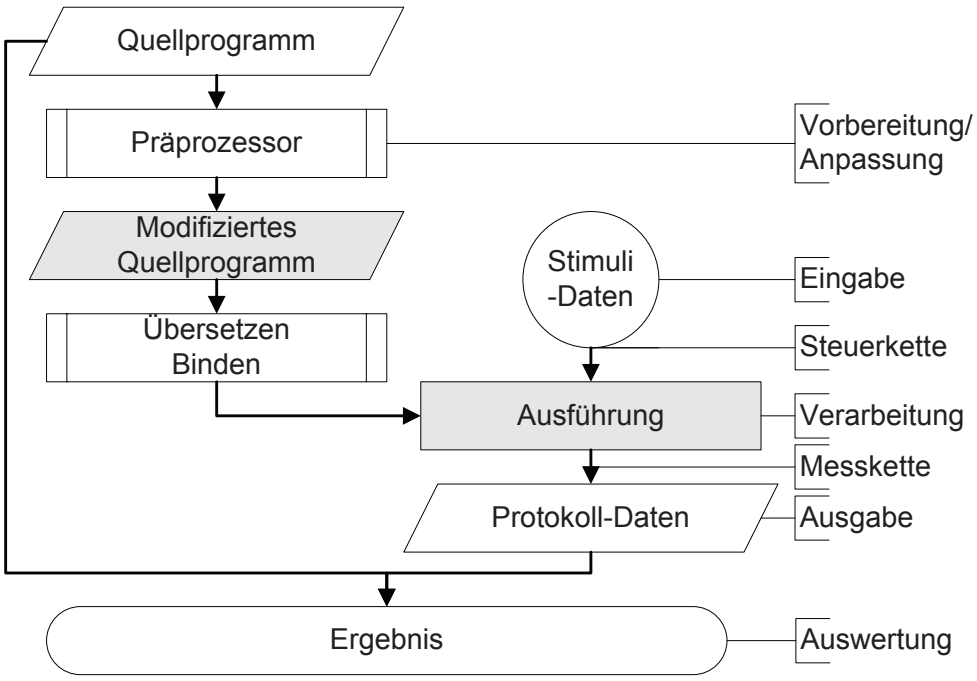


Abb. 3.1: Ablauf der Laufzeitmessung

In den nächsten Abschnitten werden diese drei Schritte im Detail erläutert. Zum besseren Verständnis werden die Schritte eins und drei zuerst diskutiert und der Schritt zwei in die Teile Laufzeitmessung an der Zielarchitektur und an der Nachbildung der Zielarchitektur unterteilt. Auf Messung basierende Verfahren sind den dynamischen Software-Testverfahren zuzuordnen, sie stellen eine Stichprobenanalyse[1] dar. Die Messung wird entweder am Zielsystem oder an der Nachbildung des Zielsystems durchgeführt.

3.1.1 Instrumentierung des Programms

Die Instrumentierung erfolgt durch eine Erweiterung des Programmcodes, hierfür werden Analysemarken, das sind zusätzliche Befehle, in das Programm eingefügt. Diese Erweiterungen ermöglichen es, bei der Ausführung des kompilierten Programms, eine Signalisierung über den gerade abgearbeiteten Abschnitt auszugeben. Über die Zeitdifferenz zwischen zwei ausgeführten Analysemarken kann die Laufzeit des dazwischen liegenden Programmabschnittes durch eine Zeitmessung bestimmt werden. Die Instrumentierungsmöglichkeiten eines Programms sind die Erweiterung:

1. des Quelltextes um zusätzliche Befehle [48],
2. durch Compiler-Optionen, die (Profiler)-Marken einfügen [46, 47],
3. durch Veränderung des bereits compilierten Programms durch das Einfügen von Analysemarken [73] und
4. durch die Instrumentierung des Programmaufrufes, bei dem das aufgerufene Programm unverändert bleibt.

Die Laufzeiteinflüsse von instrumentierenden Verfahren hängen im Wesentlichen von der Architektur des Zielsystems und von dem messtechnischen Verfahren ab, das die Signale der Instrumentierung aufnehmen kann. Verfahren, die eine automatisierte Instrumentierung auf Basis des Produktcodes vornehmen, d. h., der Quellcode wird eingelesen und in einem weiteren Schritt instrumentiert, werden in [48, 49] beschrieben.

Bewertung: Der Vorteil ist, dass diese Methode im Prinzip wie eine Stoppuhr gesteuert werden kann. Die Zeitmessung kann an beliebigen Stellen im Quellcode gestartet und gestoppt werden und ist bei richtiger Anwendung sehr genau. Nachteilig ist, dass die Instrumentierung des Quellcodes – also die eingefügten Befehle – selbst Rechnerressourcen belegen, d. h., es müssen mehr Befehle ausgeführt werden als beim unmodifizierten Quelltext. Allgemein spricht man vom sogenannten „Mess-Overhead“ oder kurz Overhead. Je kürzer die Zeit zwischen zwei Instrumentierungspunkten ist, desto stärker ist der Einfluss des Mess-Overheads auf die wahre Laufzeit. Bei komplexen Hardwarearchitekturen ergibt sich zusätzlich zur unmittelbaren Verzögerung durch die Ausführung der Instrumentierung eine Veränderung der Belegung des Caches und der Pipeline, die das Laufzeitverhalten zusätzlich beeinflusst. Die durch die Instrumentierung benötigte Laufzeit muss bei der Laufzeitbestimmung als Korrektion berücksichtigt werden. Die Schwierigkeit liegt bei Hardwarearchitekturen, die Cache, Pipelining oder andere Beschleunigungsmechanismen aufweisen. Hier entstehen neben der bekannten systematischen Messabweichung auch unbekannte und zufällige Messabweichungen, die nicht korrigiert werden können. Eine Auseinandersetzung mit dem Thema Messabweichung erfolgt in der Literatur bisher nur bis zur Beschreibung des Mess-Overheads, der die bekannte systematische Messabweichung angibt. Ein Umgang mit dem unbekannten und zufälligen Anteil der Messabweichung wird nicht beschrieben, wodurch eine Abschätzung der Genauigkeit der Verfahren nur eingeschränkt möglich ist.

[1] Analyse einer Teilmenge aus der Gesamtheit gleichartiger Einheiten [20]

3.1.2 Laufzeitermittlung aus Profiling-Informationen

Beim Profiling erfolgt die Analyse des Laufzeitverhaltens aus den Profiling-Informationen. Dabei wird die Anzahl von Aufrufen und Durchläufen von Funktionen, Grundblöcken oder Befehlen gemessen. Ermöglicht wird dies durch Compilererweiterungen oder durch Betriebssystemfunktionen. Beim Kompilieren des Programms werden Profiling-Marken in die Objektdatei geschrieben und Standard-Bibliotheken in die ausführbare Datei eingebunden, die das Profiling [45] aktivieren. Nach dem Ausführen des Programms werden die Profiling-Informationen für den Programmlauf ausgegeben. Darüber hinaus können nebenläufige Prozesse und Speicherzugriffe verfolgt werden. Der Programmcode ist mit Markern instrumentiert, die signalisieren, welcher Programmabschnitt gerade abgearbeitet wird. Die Laufzeiten, die ausgegeben werden, basieren auf einem Samplingverfahren oder auf der Instrumentierung [51] und sind in ihrer Genauigkeit abhängig vom verwendeten Verfahren zu unterscheiden.

Beim Samplingverfahren erfolgt keine exakte Auswertung aller Befehle, die abgearbeitet werden, sondern es wird stichprobenartig eine Auswertung der Programmaktivitäten vorgenommen. Diese Funktionalität muss vom Betriebssystem unterstützt werden. Die Messdaten werden dann einer statistischen Auswertung unterzogen. Die instrumentierenden Verfahren haben direkte Rückwirkung auf das zu messende Programm, zum einen durch die Veränderung der Ablaufumgebung (s. Abschn. 3.1.1), zum anderen wird durch den Einfluss der Messkette zur Ausgabe der Messdaten die Laufzeit des Programms beeinflusst. In den folgenden Abschnitten werden die Messketten beschrieben, die eine Ausgabe der Messdaten ermöglichen.

Eine Alternative zum Profiling bietet das Tracing (Ablaufverfolgung). Beim Tracing erfolgt eine Verfolgung des Programmablaufes. Ein Trace bildet eine Anweisungssequenz aus den ausgeführten Befehlen, in dem die Programmzweige den durchlaufenen Programmpfad markieren.

Laufzeitberechnung auf Basis der Ablaufdaten

Die eigentliche Laufzeitbestimmung erfolgt, wie in dem in [50] beschriebenen Verfahren, auf Basis der Ablaufdaten. Ist bekannt, wie lang die Dauer t_i jedes Befehls ist, kann die Gesamtdauer des Programms Δt bestimmt werden. Die Berechnung erfolgt nach Gl. (3.1) aus den in der Ablaufverfolgung protokollierten Grundblöcken bzw. Befehlen n_i.

$$\Delta t = \sum n_i \cdot t_i \tag{3.1}$$

Bewertung: Dieses Verfahren wird von einer Reihe von Werkzeugen unterstützt und ermöglicht einen hohen Automatisierungsgrad. Der Nachteil ist jedoch, dass Cache und Pipelining bei der Berechung der Laufzeit aus den Ablaufdaten nicht berücksichtigt werden. Dies führt bei der Laufzeitbetrachtung zu großen Abweichungen, da die Laufzeiten der Befehle und Blöcke aufgrund von Cache und Pipelining nicht als konstant angenommen werden können.

3.1.3 Laufzeitmessung an der Zielarchitektur

Abhören und Aufzeichnen der Bussignale

Ein Verfahren, das ohne Instrumentierung auskommt, ist die Messung der Bussignale an der Zielhardware und wird in [59, 60] beschrieben. Hierbei werden während der Ausführung des Programms die Bussignale über einen Logikanalysator (Logic State Analyser, LSA) protokol-

liert. Der LSA nimmt dabei die ablaufenden Adressen direkt an der Verdrahtung der Zielhardware auf. Diese werden in einen Tracespeicher bzw. Ereignisspeicher geschrieben, die Sequenz der aufgezeichneten Adressen entspricht dem Takt ihrer Ausführung. Die Bestimmung der Laufzeit erfolgt über die benötigten Taktzyklen zwischen zwei definierten Adressen. Die Bestimmung des zugehörigen Programmpfades erfolgt aus den aufgezeichneten Tracedaten, aus denen sich der Assemblercode des Programms rekonstruieren lässt.

Bewertung: Dieses Verfahren bietet eine sehr genaue Laufzeitbestimmung, wenn eine Hardwarearchitektur verwendet wird, die keinen Cache oder Pufferspeicher verwendet. Durch diese Speicher werden Bussignale verschleiert, die nicht am externen Bus anliegen, d. h., direkte Zugriffe auf den Cache und damit die Abarbeitung von gecachten Programmteilen sind nicht sichtbar. Ein weiterer Nachteil liegt darin, dass für Derivate der aktuellen Mikrocontroller für Steuergeräte kein externer Bus verfügbar ist.

Abhören und Aufzeichnen externer Ports

Eine andere Variante zum Abhören und Aufzeichnen der Bussignale ist die Möglichkeit, während der Ausführung einen freien Port zu beschreiben (s. [78]) und diesen simultan mit Hilfe von geeigneter Hardware wie dem LSA auszulesen. Hierfür muss das Programm mit entsprechenden Schreib-Befehlen instrumentiert werden. Die Laufzeit ist die Zeit zwischen den Aufrufen der Analysepunkte, die dann über eine externe Zeitmessung durch Auswertung der Port Aktivitäten bestimmt werden kann.

Bewertung: Die Genauigkeit der Laufzeitbestimmung hängt von der Auflösung des LSA und dem Einfluss der Analysepunkte auf das Programm ab. Da bei aktuellen Motorsteuergeräten meist keine ungenutzten Ports verfügbar sind, ist eine Verwendung dieses Verfahrens zur Laufzeitbestimmung nicht geeignet.

Abfragen eines Hardware-Timers

Ein weiterer Ansatz ist es, statt einer externen Zeitmessung einen internen Zeitbaustein des Zielsystems zu nutzen. Petters stellt diese Methode in [57] für Prozessoren (Intel-Pentium) mit Cycle Counter vor, die Firma Infineon stellt in [58] ein ähnliches Verfahren für den TriCore Prozessor auf Basis des System-Timer vor. Die Laufzeitmessung erfolgt durch das Abfragen eines Hardware-Timers vor und nach der Ausführung des zu messenden Programmcode-Segments. Die System-Timer zählen die Taktzyklen seit dem Start des Systems. Durch Bildung der Differenz aus den Timerwerten kann die Anzahl der Taktzyklen (Timerzyklen) zur Abarbeitung des Segmentes bestimmt werden.

Eine weitere Variation des Vorgehens bieten prozessorinterne Zyklenzähler (Cycle Counter), die am Segmentanfang gestartet und am Segmentende gestoppt und ausgelesen werden. Die Taktzyklen lassen sich dann in Abhängigkeit von der Taktung und Synchronisierung zum Prozessor in eine Zeiteinheit umrechnen.

Bewertung: Die Genauigkeit der Laufzeitbestimmung hängt von der Auflösung des Systemtimers und dem Einfluss der Analysepunkte auf das Programm ab. Die aktuelle Steuergerätegeneration verfügt über einen Systemtimer, eine Verwendung dieses Verfahrens ist möglich. Der hohe Analyseaufwand und eine fehlende Abbildung der gemessenen Laufzeit auf den durchlaufenen Programmpfad sprechen gegen eine Verwendung dieser Methode. Dagegen sind internen Zyklenzähler nicht in allen Prozessoren vorhanden, eine allgemein verfügbare Lösung kann damit nicht umgesetzt werden.

3.1.4 Laufzeitmessung an der Nachbildung der Zielarchitektur

Hardwaresimulator

Im Gegensatz zu Simulatoren, die nur die Software (also den Algorithmus) testen, kann mittels eines Hardwaresimulators das dynamische Verhalten eines Rechnersystems mit Prozessor, Cache, Speicher usw. nachgebildet werden. Die Software wird nicht auf dem Zielsystem ausgeführt, sondern auf einem Entwicklungssystem. Der zeitliche Ablauf eines Programms kann so nachgestellt und ein Trace ausgeben werden.

Bewertung: Die Genauigkeit der Laufzeitbestimmung hängt von der Qualität der Nachbildung des Mikrocontrollers ab. Niedriger Analyseaufwand des Trace zur Laufzeitbestimmung und eine Abbildung der Laufzeiten auf den Programmpfad sprechen für eine Verwendung dieser Methode. Meist werden jedoch Prozessordetails nicht veröffentlicht. Hardwaresimulatoren sind zudem nicht für alle Steuergeräteprozessoren verfügbar, eine allgemeine Lösung kann damit nicht umgesetzt werden.

In-Circuit-Emulatoren

Ein anderes Vorgehen ermöglicht der Einsatz eines In-Circuit-Emulators (ICE), der den eigentlichen Prozessor auf dem Zielsystem ersetzt. Der Emulator stellt zusätzliche Analysefunktionen bereit [64], es werden Signale herausführt, um den Programmablauf verfolgen zu können. Realisiert werden solche Emulatoren als Bond-Out Chip, FPGA oder als Logikschaltungen, die den Controller komplett nachbilden. Eine identische Nachbildung wird meist aus wirtschaftlichen oder technischen Gründen nicht umgesetzt. Eingesetzt werden ICE vor allem beim Debugging, da die Registerinhalte und der Programmablauf analysiert werden können. Darüber hinaus enthalten ICE Tracespeicher (interne Speicher) zur Aufzeichnung von Bus und Portsignalen, um die Laufzeiten und den Programmpfad zu bestimmen. Die Software läuft im Zielsystem ohne Zeiteinschränkungen und zusätzlich stehen alle Debug-Möglichkeiten zur Verfügung.

Bewertung: Die Genauigkeit der Laufzeitbestimmung hängt von der Nachbildung des Controllers ab. Bei ICE, die einen Prozessor gleichen Typs mit zusätzlichen Debugger-Funktionen bereitstellen, ist der Einfluss der zusätzlichen Funktionen bei der Laufzeit zu beachten. Eine Verwendung dieses Verfahrens zusammen mit dem Funktionstest ist möglich. Niedriger Analyseaufwand und eine Abbildung der Laufzeiten auf den durchlaufenen Programmpfad sprechen für eine Verwendung dieser Methode. Der ICE ist jedoch nicht für alle Steuergeräte vorhanden, eine allgemein verfügbare Lösung kann damit nicht umgesetzt werden.

3.1.5 Dynamische Verfahren

RTA-TRACE aus der ETAS – Toolkette

RTA-TRACE ist ein Software Logicanalyzer zur Laufzeitmessung, der es erlaubt, das ausgeführte Programm zur Laufzeit zu beobachten. RTA-TRACE ermöglicht neben einer Laufzeitanalyse von Tasks und Interrupt-Service-Routinen auch die Stack- und CPU-Auslastung zu ermitteln. Je nach Betriebssystem der Zielarchitektur ist eine Analyse auf Prozessebene möglich [61]. Es setzt auf einem instrumentierten Betriebssystem auf.

Bewertung: Der Anwender erhält eine graphische Auswertung der Messergebnisse. Nachteilig ist, dass RTA-TRACE keine Pfadanalyse durchführt, hier wird deutlich, dass es sich von seiner Zielsetzung her relativ stark von den anderen in dieser Arbeit vorgestellten Tools unterscheidet, stattdessen arbeitet es in höchster Auflösung auf Prozessebene, wodurch eine Un-

sicherheit bezüglich der Pfad- oder Zweigabdeckung der Laufzeitanalyse nicht vermieden werden kann. Es kommt hinzu, dass die Instrumentierung durch ihre Ausführung auf dem Zielsystem die Laufzeit beeinflussen kann. Der Benutzer muss den WCET-Pfad über die Eingangsdaten vorgeben. Das Verfahren liefert von vornherein keine Hinweise auf den WCET-Pfad oder die WCET.

Messverfahren der TU Wien

An der Technischen Universität Wien wurden mehrere Verfahren zur Laufzeitmessung als Forschungsprototypen implementiert. Das Messverfahren der TU Wien zeichnet sich dadurch aus, dass es benötigte Eingangsdaten selbstständig generieren kann. Dazu wendet es genetische Algorithmen an. Das Messverfahren setzt nach Puschner in [63] auf eine Messumgebung mit externer Zeitmessung auf, es erfolgt das Abhören und Aufzeichnen externer Signale des Zielsystems. Die Untersuchung läuft dann wie folgt ab. Zuerst wird das zu untersuchende Programm mit zufällig generierten Eingangsdaten ausgeführt. Die gemessenen Laufzeiten der Programmsegmente werden dann als Fitness Werte des Eingangsdatensatzes aufgefasst, wobei höhere Laufzeit für höhere Fitness steht. Anhand der Fitness Werte und Eingangsdaten werden mittels genetischer Algorithmen immer wieder neue Eingangsdaten generiert mit dem Ziel, die Laufzeit zu maximieren [62]. Eine Weiterentwicklung des Verfahrens erfolgt durch Messung auf einem zyklengenauen Simulator in [65]. Das Tool unterstützt PowerPC Prozessoren und den 16-Bit Microcontroller C167 von Infineon.

Bewertung: Ein großer Vorteil des Tools ist, dass der Benutzer keine Eingangsdaten liefern muss. Nachteilig ist jedoch, dass keine Pfadabdeckung sichergestellt oder auch nur ausgegeben werden kann. Ferner weist das Tool die generell einem Messverfahren innewohnende Unsicherheiten auf.

Messverfahren von M. Petters

Petters stellt in [57] ein Path-Analysing-Tool (PAN) vor. Er benutzt dabei den Kontrollflussgraphen, den der Compiler ausgibt. Es erfolgt eine Instrumentierung für die einzelnen Programmpfade zur Laufzeitmessung, die Messung erfolgt strukturorientiert anhand des Kontrollflusses. Die Laufzeiten werden durch Auslesen des Time-Stamp Counters [55] gemessen, die verwendeten Prozessoren sind Intel-Pentium und AMD-Athlon. Die Ausführungszeiten auf Prozessoren mit Cache, Pipelines und Branch Prediction können durch Messung der Laufzeiten mit berücksichtigt werden. Die durch die Instrumentierung hervorgerufenen Messabweichungen werden allgemein als Overhead behandelt (s. Abschn. 3.1.1).

Bewertung: Der Vorteil ist, dass die Messpunkte so aufgebaut sind, dass sie nach Angaben des Autors eine genaue Messung der Laufzeiten von Programmsegmenten ermöglichen, es wird ein Vertrauensintervall angegeben. Hierfür wird der Eintritt in die Messroutine serialisiert und Hardwarearchitekturmerkmale für den Worst-Case-Fall manipuliert. Des Weiteren werden Anhaltspunkte genannt, die durch den Einfluss der Messung zu berücksichtigen sind, eine genaue Untersuchung von systematischen und zufälligen Messabweichungen erfolgt nicht. Nachteilig ist auch, dass keine Pfadabdeckung sichergestellt oder auch nur ausgegeben werden kann.

Anwendung findet das Verfahren in der Arbeit von Bülow [66], hier wird die Messroutine auch für Multiprozessorsysteme dargestellt. Eine Zusammenstellung von Methoden, Verfahren und Tools zur Laufzeitbestimmung ist in [62] und [67] zu finden. Hier werden die Verfahren den einzelnen Forschungsgruppen entsprechend dargestellt und bewertet.

3.2 Statische Prüfmethoden zur Laufzeitbestimmung

Im Gegensatz zu den dynamischen Prüfmethoden wird das Programm bei statischen Verfahren nicht ausgeführt. Die statische Analyse lässt sich in drei logische Abschnitte untergliedern: Softwareanalyse, Architekturanalyse und Berechnung der Laufzeitgrenzen. Der Vorteil gegenüber einer Messung ist die Möglichkeit, eine garantierte WCET-Obergrenze formal zu ermitteln. Diese stellt jedoch immer eine Übererwartung dar und verschenkt damit Performance zugunsten der höheren Sicherheit. Die statische Analyse erfordert außerdem keine Eingangsdaten (sog. Stimuli), die bei einem Messverfahren vorhanden sein müssen, benötigt dafür aber oftmals die Hilfe des Benutzers (z. B. in Form von Annotationen, also Kommentaren im Quellcode), um den Programmablauf vollständig nachzuvollziehen.

3.2.1 Softwareanalyse

Die Softwareanalyse hat die Aufgabe, mögliche Ausführungspfade durch das Programm zu ermitteln. In der Literatur (z. B. [3]) wird diese Analyse häufig auch als Flussanalyse oder Kontrollflussanalyse bezeichnet, was als Oberbegriff für diese in verschiedenen Tools mit verschiedenen Arbeitsschritten realisierte Analysephase jedoch unzureichend erscheint (auch wenn es im konkreten Fall zutreffen mag). Stattdessen wird hier der allgemeinere Begriff Softwareanalyse verwendet, der einerseits die Kontrollflussanalyse einschließt und andererseits eine saubere Trennung für alle existierenden Methoden und Tools ermöglicht. Die im Folgenden aufgeführten Methoden sind keinesfalls alle in einem Tool implementiert oder Bestandteil eines einzigen Ansatzes. Ihre Notwendigkeit leitet sich im Wesentlichen von der Komplexität der verwendeten Architektur und des untersuchten Programms ab. Je nach Programm und Hardware sind die Aufgaben der Softwareanalyse:

- Aufbau eines Kontrollflussgraphen,
- Wertebereichsanalyse für lokale Variablen und CPU Register,
- Bestimmung der Schleifenbegrenzungen (max. Schleifeniterationen),
- Analyse der Eingangsdatenabhängigkeiten und
- Finden von sogenannten unmöglichen oder falschen Programmpfaden (Infeasible Paths), also Pfaden im Kontrollfluss, die im Programmablauf nicht beschritten werden können.

Definition 3.1 ausführbarer und nichtausführbarer Programmpfad [2]**:** Ein ausführbarer Programmpfad ist ein Programmpfad p, $p \in P$, der bei mindestens einer Eingabe durchlaufen wird. Man kann den ausführbaren Programmpfad auch Ausführungspfad ap, $ap \in AP \subseteq P$, oder Trace nennen. Die nichtausführbaren Programmpfade werden als falsche oder unmögliche Programmpfade fp, $fp \in FP \subset P$ und $FP \cap AP = \varnothing$, bezeichnet.

Hier sind AP die Menge aller ap und FP die Menge aller fp.

Der Kontrollflussgraph steht bei der Softwareanalyse an erster Stelle, da die weiteren Analyseschritte teilweise auf dessen Kenntnis aufbauen. Um den Kontrollflussgraphen aufzubauen, partitioniert und strukturiert man im Wesentlichen den Programmcode. Dabei wird der Code zuerst in Grundblöcke aufgeteilt, die rein sequenziell abgearbeitet werden können. Zwischen diesen sequenziellen Programmteilen sind Verzweigungen angeordnet, die in C beispielsweise durch »If«-Abfragen oder Fragezeichenoperatoren entstehen können. Außerdem werden Schleifen (»while / loop«) gesucht und dargestellt. Ein Beispiel gibt Abb. 3.2.

Im Anschluss müssen Wertebereiche und Schleifengrenzen ermittelt werden. Die Ermittlung der Schleifengrenzen ist ein komplexes Problem, das nicht auf eine allgemeine Art und Weise gelöst werden kann. Das Problem ist hier, zu bestimmen, ob ein bestimmtes Programm für gegebene Eingabewerte hält bzw. terminiert [79, 80].

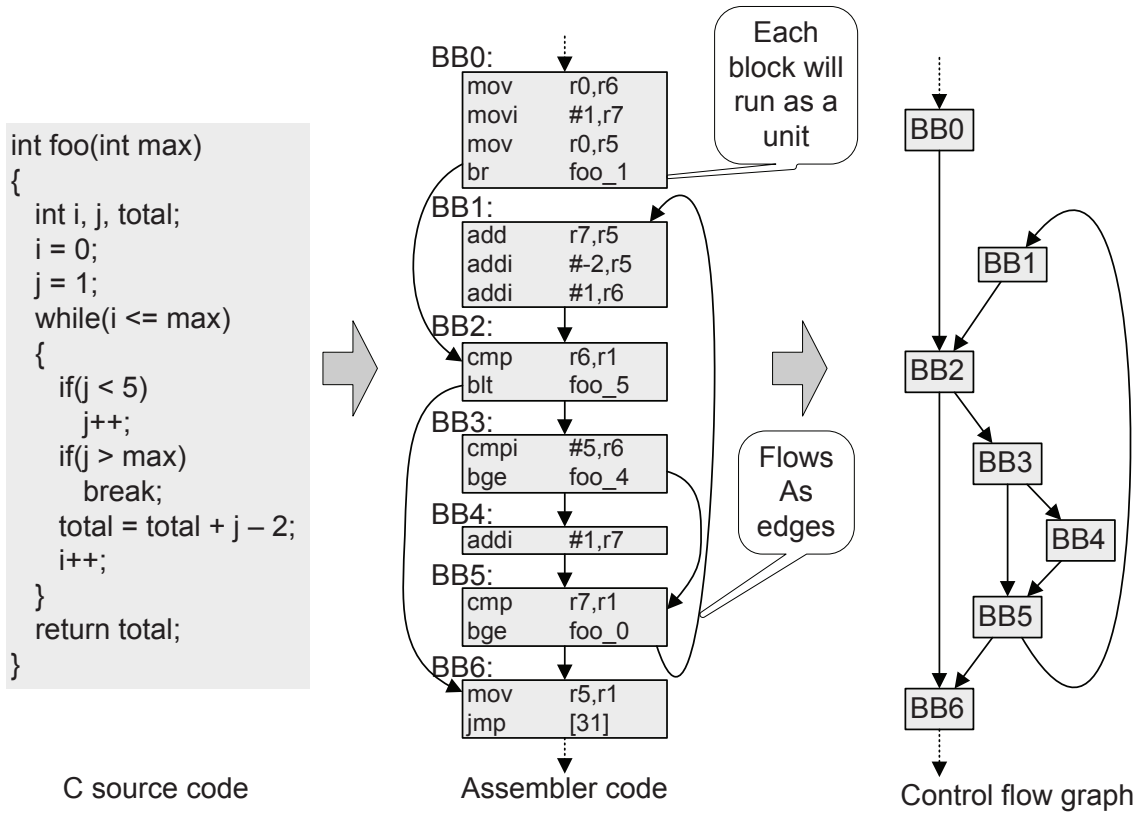


Abb. 3.2: Vom C-Code zum Kontrollflussgraphen [67]

Durch die Anforderungen, die an ein Echtzeitsystem gestellt werden, sind die zu überprüfenden Programme so zu programmieren, dass sie in jedem Fall terminieren (keine Rekursionen, explizite Schleifengrenzen). Dadurch können die Schleifengrenzen während der Analyse auf eine sichere Obergrenze festgelegt werden. Die Wertebereichsanalyse hat die Aufgabe, auf statischer Basis mögliche Werte von Variablen und CPU-Registern zu bestimmen. Durch Ermittlung von Adressbereichen hilft sie bei einer potentiellen Cacheanalyse eine Aussage über den Cachestatus treffen zu können. Außerdem sind die ermittelten Werte wesentlich für die Ermittlung der Schleifeniterationen und um unmögliche Pfade auszumachen.

Ein unmöglicher Pfad entsteht beispielsweise durch den Zweig einer Kausalbedingung, der für bestimmte Wertebereiche einer Variablen nicht vom Programmablauf beschritten werden kann. Eine Information über die Wertebereiche von Registern und Variablen kann gewonnen werden, indem der Inhalt der Prozessorregister modelliert und an jedem Programmpunkt untersucht wird. Gängige Methoden für diesen Zweck basieren entweder auf einer instruktionsweisen Modellierung des Programmablaufes im Konkreten mit dem Nachteil einer gewissen Unsicherheit, da konkrete Werte oftmals nicht vorhanden sind, oder aber auf abstrakter Interpretation. Die Zielsetzung der abstrakten Interpretation ist, das (automatisierte) Nachvollziehen des beobachteten Programms zu vereinfachen. Wie der Name schon andeutet, wird dafür geschickt abstrahiert (es werden also Informationen vernachlässigt). Dabei werden konkrete Werte zu abstrakten Werten zusammengefasst, es ist aber auch möglich, konkrete Mengen zu abstrakten Mengen zusammenzufassen und dementsprechend geschickt Wertebereiche für Variablen und Schleifenbegrenzungen zu ermitteln.

In [68] wird ein Ansatz beschrieben, der auf Intermediate-Code Ebene (ein vom Compiler erzeugter Zwischencode) arbeitet und Schleifen syntaktisch (anhand ihrer Form und Struktur) im Code erkennt. Die Begrenzungen werden dann ermittelt und die nun definierten Schleifen in einfachere Konstrukte überführt, um die Komplexität der Analyse zu reduzieren. Die übri-

gen Schleifen werden erst dann, also nachgeschaltet, mittels abstrakter Interpretation gefunden. Der Vorteil der Arbeit auf der Intermediate-Code Ebene ist, dass einerseits Optimierungen schon enthalten sind und andererseits der Code weder für die Zielhardware noch für die im Quellcode verwendete Hochsprache spezifisch ist. Dadurch wird vor allem eine hohe Portabilität erreicht und man versucht Problemen durch Optimierungen des Compilers aus dem Weg zu gehen.

Generell handelt es sich bei der Auswahl der verwendeten Codeebene in jedem Ansatz um einen Zielkonflikt, für den ein Kompromiss ausgewählt wird. Für die reine Interpretation, also für das automatisierte Verständnis, ist es sinnvoll auf der Quellcodeebene, also in der Hochsprache, zu analysieren. Allerdings kann dieser Quellcode durch den Compiler stark optimiert werden und dementsprechend stimmen untersuchter Code (nicht-optimierter Quellcode) und eingesetzter Code (Objekt) nicht überein. Da eine sinnvolle Architekturanalyse immer mit Objektcode arbeitet, ist es außerdem schwierig die notwendige Beziehung zwischen Architekturanalyse und Softwareanalyse herzustellen, also eine Aussage darüber zu treffen, welcher Teil des Objektcodes zu welchem Teil des Quellcodes gehört. Versucht man dieses Problem zu umgehen, indem man die Optimierung des Compilers deaktiviert (falls dies überhaupt möglich ist), handelt man kontraproduktiv: Es würde Performance verschenkt, was durch die WCET Analyse eigentlich vermieden werden sollte. Entscheidet sich ein Analysetool als Implementierung einer Methode also für die Softwareanalyse am Quellcode, muss es eine Möglichkeit liefern, die Compileroptimierungen zu berücksichtigen.

Eine Softwareanalyse auf Objektebene dagegen ist extrem nahe an der Hardware, spezifisch für die verwendete Architektur und macht es schwierig, spezielle Konstrukte wie Schleifen und Verzweigungen sicher zu erkennen (diese können je nach Compiler unterschiedlich umgesetzt werden). Der Vorteil liegt darin, dass der getestete Code und der im Zielsystem ausgeführte Code identisch sind. Wenn Benutzereingaben erforderlich sind, um die Analyse besser oder überhaupt erst möglich zu machen, ist es für den Benutzer, sogar für den Programmierer, nur schwer möglich diese am Maschinencode zu annotieren. Eine Beziehung zurück zum Quellcode muss dann hergestellt und eine Möglichkeit geschaffen werden, eine separate Annotation (getrennt vom Maschinencode) vorzunehmen.

Das Ergebnis der Softwareanalyse gibt Auskunft darüber, wie das untersuchte Programm abläuft und stellt die Grundlage für die Architekturanalyse und die WCET Berechnung.

3.2.2 Architekturanalyse

Der nächste Schritt einer statischen Softwareanalyse ist die Untersuchung der Architektur. Dieser Schritt wird in der Literatur (z. B. [62]) oftmals als Prozessor-Verhaltensanalyse bezeichnet. Auch dieser Begriff scheint, ähnlich wie schon der Begriff „Kontrollflussanalyse“, zu speziell, um die tatsächlich notwendigen Abläufe zu erfassen, und damit ungeeignet. Stattdessen wird in dieser Arbeit verallgemeinernd der Begriff „Architekturanalyse“ verwendet. Der Beweggrund dafür ist, dass diverse Architekturmerkmale wie Cache und Speicher in dieser Analyse berücksichtigt werden müssen und diese Merkmale nicht zwingend auf dem Prozessor integriert sind. Die Aufgaben der Architekturanalyse sind:

- Ausführungszeiten für Programmteile zu ermitteln und
- Kontext der Prozessor- und Peripheriezustände zu ermitteln.

Die Architekturanalyse benötigt das Ergebnis der Softwareanalyse als Eingangsdaten. Sie muss auf der Objektebene stattfinden, da nur das fertige Programm alle nötigen Informationen für diese Analyse enthält. Außerdem ist die Architekturanalyse – bedingt durch ihre Aufgabe – stark hardwareabhängig und benötigt daher ein Modell der Zielarchitektur, welches auch die speziellen Features der Architektur berücksichtigt. Für die hier favorisierte Architektur des

TriCore 1.3 [10, 11] bedeutet das beispielweise, dass Pipeline und Cachezustand und die in Kombination damit potentiell zu Anomalien führenden, superskalaren Features Branch Prediction und spekulative Ausführung sowie out-of-order Execution (die Fähigkeit des Prozessors, die Ausführungsreihenfolge von Befehlen dynamisch anzupassen, um die Pipelineauslastung zu optimieren) mit berücksichtigt werden müssen.

Als Laufzeitanomalie bezeichnet man den kontraintuitiven Einfluss der lokalen Laufzeit einer Instruktion auf die globale Laufzeit einer Task. Das bedeutet beispielsweise, dass ein Worst-Case auf Instruktionsebene durch die entstehende Verzögerung verhindert, dass später eine noch größere Verzögerung eintritt. Abbildung 3.3 zeigt, wie ein Cache-Hit dazu führen kann, dass eine falsche Verzweigungsvorhersage zu einer spekulativen Ausführung führt, deren Auswirkungen auf Variablen, Cache und Pipeline später rückgängig gemacht werden müssen. Der gesamte Zeitverlust ist dadurch größer als ein Cache-Miss [62]. Das Problem besteht hierbei nicht unbedingt in der Verzögerung selbst. Diese kann in Kauf genommen werden, da die Anzahl richtiger Zweigvorhersagen in der Regel überwiegt und die Architektur eine falsche Vorhersage im weiteren Programmablauf mehr als ausgleicht. Vielmehr bedeutet die Existenz dieser Anomalien für die Architekturanalyse erheblichen Mehraufwand, da die gesamte Ausführungshistorie zu jedem Programmpunkt gespeichert werden muss. Wenn die Analyse eine Instruktion erreicht, zu der es keinen eindeutigen, durch den vorhergegangenen Programmverlauf erreichten Prozessorzustand gibt, müssen alle Zustände betrachtet werden; es ist nicht möglich im Vorhinein einen dieser Zustände als den Worst-Case auszumachen.

Durchgeführt wird die Architekturanalyse auf einem abstrakten Modell der Zielarchitektur. Ein Problem der Architekturanalyse ist eben dieses Modell zu implementieren, da komplexe Architekturen vom Hersteller nicht vollständig dokumentiert werden. Cacheinhalt und Pipelinestatus können bei komplexen Zielarchitekturen am besten durch abstrakte Interpretation ermittelt werden, eine detaillierte Erklärung zur Cacheanalyse bietet [69]. Im Prinzip werden Cache-Zugriffe dabei klassifiziert als „always hit“ (durch die Ausführungshistorie in jedem Fall im Cache), „always miss“ (nie im Cache) sowie „persistent“ (Cachestatus für ersten Zugriff unbekannt, jeder folgende Zugriff ist ein Treffer, z.B. bei Schleifen) und „not classified“ (Cachestatus unbekannt). Diese Klassifizierung wird erreicht durch geschickte Kombination der Ausführungshistorie, also aller Wege, die zu dem aktuellen Programmpunkt führen können.

Im Hinblick auf den vorangegangenen Absatz lässt sich am Beispiel der Cacheanalyse folgende Aussage machen: Wenn der Cachezustand „always hit“ oder „always miss“ ermittelt wurde, muss nur dieser Zustand an der aktuellen Stelle der Analyse betrachtet werden. Tritt einer der nicht definierten Zustände auf, müssen alle Möglichkeiten berücksichtigt werden. Das Ergebnis der Architekturanalyse ist eine Information darüber, wie das Programm auf der speziellen betrachteten Hardware läuft.

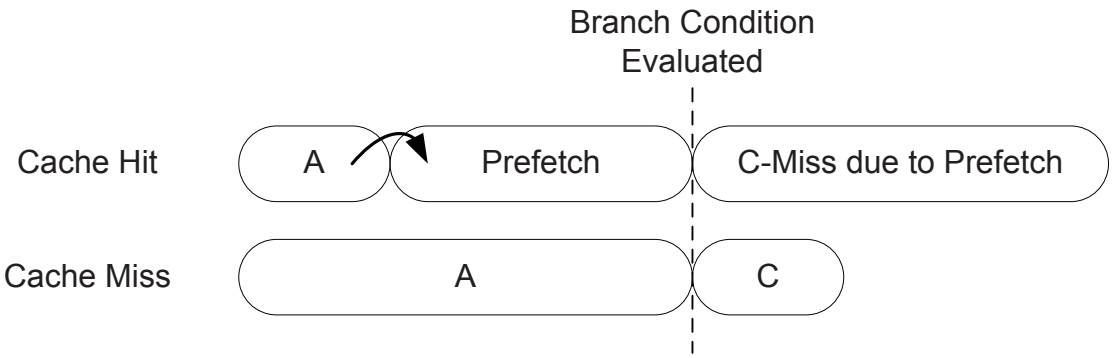


Abb. 3.3: Laufzeitanomalie: Zeitverlust durch Cache-Hit bei spekulativer Ausführung [62]

3.2.3 Auswertung der Laufzeitanalyse - Berechnung der Grenzen

Im Anschluss an die Software- und Architekturanalyse wird die Laufzeit berechnet. Dies geschieht durch Kombination des Programmablaufes aus der Softwareanalyse mit dem Laufzeitverhalten der Architektur aus der Architekturanalyse. Zur Berechnung der WCET gibt es im Wesentlichen drei Ansätze, die in Abb. 3.4 dargestellt werden. In Abb. 3.4 (a) wird der Kontrollflussgraph aus der Softwareanalyse dargestellt, der mittels der Architekturanalyse mit Laufzeitinformationen (in Zyklen) versehen ist.

Die Abb. 3.4 (d) zeigt den einfachsten Weg zur Berechnung der WCET aus Pfadinformationen und Laufzeiten. Dabei wird mit der tiefsten Verzweigung des Pfades begonnen, Knoten nach bestimmten Regeln zusammenzufassen. Beispielsweise gilt bei konditionalen Verzweigungen die Regel [Max(Unterknoten) + (Zeit für die Entscheidung selbst)] (siehe auch „Transformation rules“ in Abb. 3.4 (d) links). Der Vorteil bei dieser einfachen Methode ist, dass das eigentliche Problem relativ schnell stark vereinfacht wird und deswegen in kurzer Zeit und ohne aufwendige Algorithmen gelöst werden kann. Der Nachteil ist jedoch, dass diese Methode im Prinzip nur für Architekturen geeignet ist, bei denen die Ausführungszeiten nicht kontextabhängig sind. Um sie dennoch für moderne Architekturen einsetzen zu können, müsste man den Baum in verschiedenen Umformungen betrachten um verschiedene Ausführungshistorien zu berücksichtigen [62]. Die Abb. 3.4 (b) zeigt, wie eine pfadbasierte Laufzeitberechnung abläuft: Im Wesentlichen werden die explizit aufgezählten, möglichen Ausführungspfade herangezogen und deren Laufzeiten jeweils berechnet.

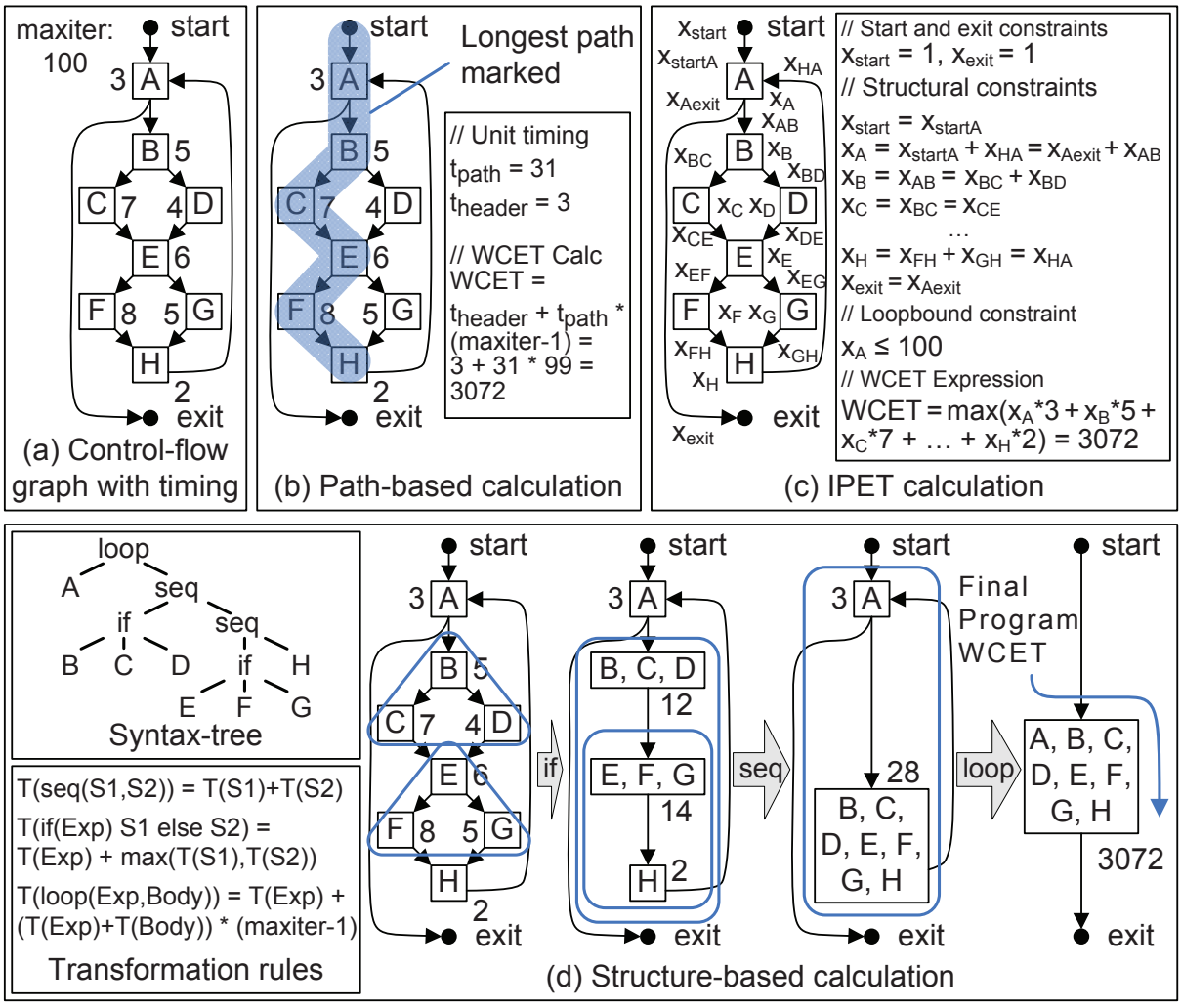


Abb. 3.4: Verschiedene Ansätze zur Laufzeitberechnung [62]

Der Pfad mit der längsten berechneten Laufzeit bestimmt die WCET der Task. Der Vorteil dieser Methode ist der enge Bezug zum tatsächlichen Programmablauf und die – bei kleinen Programmen – gute Nachvollziehbarkeit. Problematisch ist jedoch Behandlung von verschachtelten Schleifen. Diese führen im Allgemeinen zu einer sehr großen Zahl an möglichen Pfaden und schränken damit die Möglichkeiten dieser Methode so ein, dass selbst auf aktueller Hardware der Worst-Case-Pfad oftmals nur mittels heuristischer Methoden gesucht werden kann. Dann jedoch ist nicht mehr sichergestellt, dass der Worst-Case gefunden wird und damit eine sichere Grenze angegeben werden kann.

Die Abb. 3.4 (c) zeigt den IPET-Ansatz zur WCET-Bestimmung. IPET steht für „Implicit Path Enumeration Technique“, ist also eine Methode zur impliziten Pfadaufzählung. Dabei werden die einzelnen Pfade schon vorher im Rahmen der Softwareanalyse nicht mehr direkt angegeben, sondern stattdessen wird ein Gleichungssystem aufgestellt, dass die Knoten und Kanten des Kontrollflussgraphen und die Grundblocklaufzeiten sowie deren Ausführungshäufigkeiten miteinander in Beziehung setzen. Es ergibt sich eine „Kostenfunktion“ für die Anzahl der benötigten CPU-Zyklen, die mittels eines ILP-Solvers maximiert (für die WCET) und minimiert (für die BCET) werden kann.

Der Vorteil dieser Methode ist, dass die Gleichungen gut durch zusätzliche Angaben des Benutzers ergänzt werden können. Es handelt sich um eine intuitive und logische, aber trotzdem formale Art der Notation. Das Verfahren ist zwar sehr komplex und die Anzahl der Gleichungen groß, aber durch die Überführung in ein bekanntes mathematisches Problem erspart sich diese Berechnungsmethode die eigene Implementierung einer Suchmethode für den längsten Pfad und kann auf bestehende, bereits optimierte ILP-Solver zurückgreifen.

3.2.4 Sonderfall: symbolische Simulation

Lundqvist beschreibt in [70] eine weitere statische Methode zur Laufzeitbestimmung, mit der Softwareanalyse, Architekturanalyse und Berechnung der Laufzeitgrenzen gemeinsam implementiert werden können. Die Methode setzt einen modifizierten Architektursimulator ein, der in der Lage ist, das Programm ohne Eingangsdaten und mit teilweise unbekanntem Cache und Pipelinestatus auszuführen [62]. Problematisch ist an diesem Ansatz, dass für die Architektur ein entsprechender Simulator vorhanden sein bzw. erstellt werden muss und dass es für viele komplexe Zielarchitekturen nicht einmal einen Simulator gibt, der in der Lage ist, deren Verhalten mit vorhandenen Eingangsdaten nachzubilden. Der Grund dafür ist einerseits oftmals die unzureichende Dokumentation durch den Hersteller der Architektur und andererseits die komplexe Implementierung der Mikroarchitektur einer modernen CPU in die Software. Ein weiteres Problem dieses Ansatzes ist der zeitliche Aufwand, da ein Simulator oftmals mehrere Größenordnungen langsamer arbeitet als die Zielarchitektur selbst [62].

3.2.5 Statische Verfahren

Chalmers Prototyp

Der Chalmers Prototyp ist ein Forschungsprojekt von der Chalmers University of Technology aus Göteborg, Schweden. Es handelt sich um die Implementierung der von Thomas Lundqvist und Per Stenström beschriebenen Methoden, um die WCET auch auf modernen, superskalaren Prozessoren mit Cache und Pipeline zuverlässig bestimmen zu können.

Dazu haben Lundqvist und Stenström einen Ansatz entwickelt, der Pfad und Laufzeitanalyse durch zyklengenaue symbolische Ausführung integriert. Die Besonderheit des gewählten Ansatzes ist, dass diese symbolische Ausführung auch mit unbekannten Eingangsdaten mög-

lich ist. Das entwickelte Verfahren vereint also die Genauigkeit eines Simulationsansatzes mit dem Vorteil einer statischen Methode, auf konkrete Eingangsdaten verzichten zu können [62]. Während der symbolischen Ausführung werden automatisch nicht ausführbare Pfade ausgeschlossen. Bei Eingangsdatenabhängigkeiten werden alle Möglichkeiten ausgeführt, lediglich die Anzahl der Schleifeniterationen muss vom Benutzer begrenzt werden [62]. Dadurch ergibt sich eine je nach Programm sehr große Zahl an ausführbaren und damit symbolisch auszuführenden Pfaden, wodurch die Ausführungsgeschwindigkeit der Analyse sehr langsam ausfällt. Zur Geschwindigkeitssteigerung kann der Prototyp Pfade ausschließen, die nicht zur WCET führen werden. Um Pfade sicher auszuschließen, muss das Tool eine sichere Annahme treffen, ob und wie sich die Ausführung eines Pfades auf die zukünftige Performance auswirkt, also insbesondere wie Cache und Pipeline durch die Ausführung des Pfades hinterlassen würden, und welchen Einfluss dies auf die Laufzeit folgender Instruktionen haben kann.

Bewertung: Der Zielkonflikt dieses Tools besteht also vor allem darin, dass einerseits durch die symbolische Ausführung (eine Art von Simulation) die Analysedauer etwa proportional zur Programmlaufzeit auf der Zielarchitektur ist, aber – ein typisches Problem der Simulation – um einen Faktor größer. Um die Analysedauer zu reduzieren, schließt das Tool Pfade aus. Da diese aber nicht ausgeführt werden (sonst wäre das Ausschließen zwecklos!), muss es einen sicheren Architekturzustand annehmen, der erreicht worden sein könnte, wäre der Pfad ausgeführt worden. Diese „Sicherheit“ führt jedoch zu einer pessimistischeren Erwartung für die folgende symbolische Ausführung. Der Prototyp unterstützt eine mehrstufige superskalare Pipeline und Cache. Bisher wird lediglich die PowerPC Architektur unterstützt und eine Anpassung an eine andere Architektur ist als kompliziert anzusehen, weil dafür ein zyklengenauer Simulator nicht nur verfügbar sein, sondern auch entsprechend angepasst werden müsste.

aiT

Das WCET-Tool aiT ist ein kommerzielles Produkt. Es basiert auf dem Prinzip der abstrakten Interpretation, dessen Anwendung im Bereich der WCET-Analyse von den aiT Entwicklern Ferdinand und Wilhelm (Universität des Saarlandes) geprägt wurde.

Das Tool ist in der Lage, die Laufzeit von Codeabschnitten zu bestimmen, die ihm in ausführbarer Form (als Objekt) übergeben werden. aiT setzt auf der Executable-Ebene auf, weil aus dem Quellcode weder Informationen über den Status der CPU-Register noch Speicheradressen hervorgehen, die aiT für seine Werteanalyse und für die Berücksichtigung des Cache- und Speicherverhaltens benötigt [62]. Vom Benutzer müssen je nach zu analysierendem Programm Schleifenbegrenzungen und sogenannte „Flow Facts“, also Angaben über den Programmfluss z. B. bei zur Laufzeit ermittelten Sprüngen, mittels Parameterdateien eingegeben werden. Davon abgesehen kann der Benutzer optional die Wertebereiche von Variablen und Registerinhalte angeben, um das Analyseergebnis zu verbessern oder spezielle Betriebsmodi abzudecken.

Der Analyseablauf ist typisch für ein statisches WCET Tool. Zuerst wird aus dem eingegebenen Objektcode und der vom Benutzer erstellten Parameterdatei der Kontrollflussgraph konstruiert. Dieser bildet anschließend die Basis für alle weiteren Analyseschritte. Eine Werteanalyse ermittelt den Inhalt von Prozessorregistern für jeden Punkt im Programmablauf bzw. für jeden Ausführungskontext und leistet so einen entscheidenden Beitrag zur automatischen Ermittlung von Schleifengrenzen und berechneten Sprüngen. Um diese verschiedenen Zustände an verschiedenen Programmpunkten zu bestimmen, werden sichere Intervalle für die zu ermittelnden Werte gefunden, die die tatsächlich auftretenden Werte garantiert enthalten. An den einzelnen Programmpunkten werden mittels spezieller Mengenoperationen die vorhergehenden Prozessorzustände zu einem einzelnen Zustand kombiniert. Werte, die nicht bestimmt werden können, müssen durch den Benutzer eingegeben werden [62]. Hier ergibt

sich die Möglichkeit eines iterativen Vorgehens des Benutzers, der nach der Werteanalyse theoretisch nur die tatsächlich unbekannten Werte in Form von Annotationen ergänzt, um möglichst wenig potentiell fehlerbehaftete Handarbeit in die Analyse zu stecken. Im Sinne der Bestimmung enger Laufzeitgrenzen ist es aber sinnvoll, z. B. bekannte Schleifengrenzen anzugeben, weil das Tool diese überprüft und ermittelte Unstimmigkeiten ausgegeben werden.

Eine Cacheanalyse setzt auf den in der Werteanalyse gewonnenen Daten auf und klassifiziert Cachezugriffe als sichere Hits oder mögliche Misses. Die folgende Pipelineanalyse ermittelt das Verhalten der Prozessorpipeline für jeden als möglich ermittelten Ausführungskontext [62].

Wie schon bei der Werteanalyse wird die Pipeline modelliert und ihr ermittelter Zustand von jedem betrachteten Grundblock zum nächsten übergeben. Durch die abstrakte Interpretation ist es möglich, dass hier mehrere Zustände auftreten und übergeben werden müssen. Aus der Pipelineanalyse gehen die benötigten CPU-Zyklen pro Grundblock und Kontext hervor. Mit dieser Information wird während der Pfadanalyse die maximale Laufzeit (Worst-Case) per ILP ermittelt [72]. aiT ist eines der am weitesten entwickelten, statischen Analysetools. Zahlreiche Hilfsfunktionen unterstützen den Benutzer bei der Analyse, verifizieren seine Eingaben und machen ihn auf Widersprüche oder auf vom Programm getroffene Annahmen aufmerksam. Als statisches Tool und durch die Berücksichtigung von Cache und Pipeline liefert aiT (bei richtiger Bedienung) sichere Laufzeitgrenzen. Das Visualisierungstool aiSee kann den Kontrollflussgraph visualisieren und dabei Laufzeitgrenzen, den Worst-Case-Pfad und mögliche Cache- und Pipelinezustände anzeigen.

aiT benötigt für die Ermittlung der benötigten Zyklen ein CPU-Modell, welches speziell für die entsprechende Zielplattform verfügbar sein muss. Da es kein quelloffenes Tool ist, muss man sich diesbezüglich darauf verlassen, dass Support für die eingesetzte Plattform durch AbsInt integriert wird oder bereits wurde. Es ist jedoch für viele Zielarchitekturen verfügbar, unter anderem auch für die eingesetzte TriCore 1.3 Architektur.

Bewertung: Der Vorteil, eine garantierte Obergrenze zu ermitteln geht einher mit dem Nachteil, dass diese Grenze zwar sicher, aber nicht immer sehr eng sein muss. Wie alle Tools mit Annotationseingabe erwartet aiT von seinem Benutzer, dass er sich in der untersuchten Software gut auskennt. Letztendlich wird oftmals ein Software-Entwickler die Laufzeitanalyse durchführen müssen. Im Falle der automatischen Codegenerierung kann die Notwendigkeit der Annotation zu einer Einarbeitungszeit führen, in der der Benutzer den Code analysiert. Im Falle von fertigen Bibliotheksfunktionen, die oft nur als Objekt und nicht als Quellcode vorliegen, ist die Sache noch schwieriger: Hier ist einerseits der Entwickler dieser Funktion oftmals nicht vor Ort und andererseits muss zuvor geklärt werden, ob eine Dissassemblierung, wie aiT sie zu Analyse und Aufbau des Kontrollflussgraphen vornehmen muss, rechtlich vertretbar ist.

Ein speziell im Falle von Automobilherstellern als Software-Lieferant relevanter Nachteil ist, dass aiT zwar den Worst-Case-Pfad ermittelt und darstellt, jedoch nicht die zugehörigen Eingangsdaten angeben kann, da es den Code abstrakt interpretiert. Diese Eingangsdaten können zur Reproduktion des Analyseergebnisses beim Systemlieferanten oder zu einer weiteren Überprüfung mit einem Messverfahren erforderlich sein.

SWEET (SWEdish Execution Time tool)

SWEET ist ein WCET Analysetool der Mälardalen Universität in Schweden. Es unterstützt vollständig die Analyse von ANSI-C Programmen einschließlich Zeiger und Rekursion. SWEET ist in einen Forschungscompiler integriert und kann so eine Programmflussanalyse auf einer Zwischencodeebene durchführen, in der eventuelle Optimierungen des Compilers bereits stattgefunden haben [71]. Die Programmflussanalyse erfolgt bei SWEET in einem

mehrstufigen Ansatz. Zuerst erfolgt ein sogenanntes „program slicing“, welche für den Programmfluss nicht relevante Teile von der weiteren Betrachtung ausschließt. Mittels Werteanalyse und „pattern-matching“ wird ein Teil der Flussanalyse bewerkstelligt, ähnlich wie auch bei aiT. Komplizierterer Code wird durch ein spezielles Verfahren analysiert, das sich „abstrakte Ausführung“ nennt. Dabei handelt es sich im Prinzip um eine symbolische Ausführung bzw. Simulation, wie sie auch beim Chalmers Prototyp zum Einsatz kommt. Für die notwendigen Variablenwerte werden wie bei aiT mittels abstrakter Interpretation an den einzelnen Programmpunkten sichere Wertebereiche ermittelt. Schleifen werden von SWEET ausgerollt, d. h. in äquivalente Konstrukte überführt, die es ermöglichen jeden Schleifendurchlauf einzeln zu analysieren. Durch die „abstrakte Ausführung“ werden – wiederum vergleichbar mit dem Chalmers Prototyp – automatisch Schleifenbegrenzungen und nichtausführbare Pfade ermittelt.

Die Architekturanalyse von SWEET wird in zwei Phasen durchgeführt. Dabei wird zuerst im Rahmen einer Speicherzugriffsanalyse ermittelt, welche Speicherbereiche von einzelnen Instruktionen benutzt werden. Je nach Hardware wird auch eine Cacheanalyse für den Instruktionscache durchgeführt, die der Cacheanalyse von aiT stark ähnelt. Das Ergebnis der Speicheranalyse wird in eine Pipelineanalyse gespeist. Diese Pipelineanalyse basiert auf der Simulation mit einem zyklengenauen CPU-Modell. Dieses muss beeinflussbar sein, sodass die Ergebnisse der Speicheranalyse in Bezug auf genutzte Speicherbereiche und Verzweigungsvorhersagen schließlich in der Simulation berücksichtigt werden können. Laufzeiteffekte über mehrere Grundblöcke hinaus versucht SWEET durch aufeinander folgende Simulationsdurchgänge abzudecken [62]. Zur Berechnung der Laufzeitgrenzen sind in SWEET drei Ansätze implementiert. SWEET unterstützt sowohl eine pfadbasierte Berechnung wie in Abb. 3.4 (b), eine IPET Berechnung wie in Abb. 3.4 (c) oder einen hybriden Ansatz, der je nach der Programmflussinformationen lokal für den jeweiligen Programmteil Pfad- oder IPET basiert arbeitet [71]. Im Gegensatz zu anderen WCET Analysetools ist SWEET modular aufgebaut, sodass an den Schnittstellen der verschiedenen Analyseschritte in die Analyse eingegriffen werden kann.

Bewertung: Vorteilhaft an SWEET ist seine Ausrichtung auf einen Analyseablauf, der möglichst ohne Benutzereingriffe auskommt. Es kann beliebigen ANSI-C Code analysieren und enthält neben einer Werteanalyse zwei verschiedene Methoden zur Programmflussanalyse. Vor allem die abstrakte Ausführung kann nötige Benutzereingriffe auf ein Minimum reduzieren. Benutzereingaben sind laut [62] aber nötig, wenn Speicher dynamisch zugewiesen wird. Im Bereich der Steuergeräte-Software kommt dies jedoch aktuell nicht vor.

Nachteilig an SWEET ist, dass es nur Instruktionscache und keinen Datencache unterstützt [62], jedoch kommt dieser Nachteil bei der Tricore 1.3 Architektur nicht zum tragen, da diese keinen Data Cache aufweist. Allerdings unterstützt SWEET auch nur in-order Pipelines und kann somit für eine superskalare Architektur (mit Timing Anomalien) wie TriCore keine Laufzeitgarantie liefern. Diese Plattform wird folglich auch nicht unterstützt und eine Anpassung ist auf Benutzerseite nicht möglich. Es handelt sich bei SWEET um ein relativ weit fortgeschrittenes Forschungstool.

Bound-T

Das kommerzielle Tool Bound-T wurde ursprünglich entwickelt für die Verifikation von On-Board Software in Raumfahrzeugen, wird aber mittlerweile von Tidorum Ltd. aus Helsinki, Finnland auch für andere Anwendungsgebiete angeboten. Das Tool kann sowohl eine Laufzeit-Obergrenze als auch optional eine Stackverbrauch-Obergrenze einer Subroutine (einschließlich aufgerufener Funktionen) bestimmen. Es arbeitet auf der Executable Ebene – Quellcode des zu untersuchenden Programms muss nicht verfügbar sein [62].

Bound-T decodiert das eingegebene Programm, führt eine Programmflussanalyse durch und erstellt Kontrollfluss- und Call-Graph. Es bestimmt die Schleifengrenzen für erkannte Zählschleifen automatisch (Werteanalyse). Für andere Schleifen kann der Benutzer die Iterationen manuell angeben; bei Bound-T werden diese Angaben als Assertions bezeichnet. Zusätzlich können z. B. Variablenwerte mit diesen Assertions eingeschränkt werden um spezielle (Sonder-) Fälle zu betrachten oder auch einzuschließen [74]. Im Anschluss an die Analyse ermittelt Bound-T die Laufzeit für die entsprechende Architektur mittels einfacher abstrakter Interpretation [62] anhand eines Prozessormodells, das auf der zugehörigen Dokumentation beruht. Berücksichtigt werden verschiedene, unterschiedlich schnelle Speicherbereiche und in-order Pipelines, jedoch keine Branch Prediction und auch kein Cache [75]. Die Berechnung der WCET erfolgt per IPET. Die Datenausgabe erfolgt vorrangig tabellarisch in Textform (Ausgabe der benötigten Zyklen) sowie grafisch für Kontrollfluss- und Call-Graph [62].

Bewertung: Bound-T ist verglichen mit den anderen kommerziellen Tools (aiT und RapiTime) auf den ersten Blick recht übersichtlich. Es ist wenig modular, sodass eine Anpassung an eine andere Plattform nur durch den Hersteller realisierbar ist und dann eine „neue Version“ des Tools erfordert. Es wird per Kommandozeile gesteuert und hat keine graphische Benutzerschnittstelle (GUI). Komplizierte Architekturfeatures und superskalare Prozessoren werden nicht unterstützt. Angesichts der steigenden Verbreitung dieser Architekturen ist dies ein gravierender Nachteil von Bound-T. Es ist ein Closed-Source Programm, das heißt Anpassungen durch den Benutzer sind nicht möglich.

Zu den Vorteilen von Bound-T zählt die ausgesprochen gute Dokumentation der Bedienung [75], vor allem auch bezüglich der sogenannten Assertions (Benutzervorgaben über den Programmablauf). Während des Programmablaufs werden dem Benutzer Hinweise ausgegeben, die bei der Erstellung der Assertions hilfreich sein können [76].

Prototyp der TU Wien für statische Analyse

Bei diesem Forschungs-Prototypen handelt es sich um einen statischen Analyseansatz, der Programme analysiert, die in WCETC, einer definierten Teilmenge von C, geschrieben sind. Diese Teilmenge wurde um die Aufnahme von Annotationen vom Benutzer zur Flussanalyse erweitert [62]. Das Tool lässt sich so in die Matlab/Simulink Toolkette integrieren, dass alle (Fluss-) Informationen, die zur Laufzeitberechnung nötig sind, aus dem Simulink Projekt gewonnen werden können und eine manuelle Angabe von weiteren Informationen durch den Benutzer entfällt. Das Tool unterstützt außerdem sogenannte „back annotation“, was soviel bedeutet, dass die Schnittstelle mit Simulink bidirektional ausgeführt ist und auf diese Weise Laufzeitinformationen direkt in Simulink (als spezielle WCET-Blöcke) dargestellt werden können. Die WCET Berechnung dieses Tools erfolgt per IPET [62].

Bewertung: Ein großer Vorteil dieses Tools ist die durchgängige Integration in die Matlab/Simulink Umgebung. Wenn der Benutzer sich an ein vorgegebenes Blockset hält, sind keine Annotationen vom Benutzer nötig. Nur wenn der Benutzer C Code analysieren will, werden Annotationen notwendig. Nachteilig an dem Tool ist einerseits, dass lediglich Unterstützung für in-Order Pipelines und jump-cache implementiert ist und andererseits – dadurch bedingt – aus Sicht der heutigen Steuergeräteentwicklung nur weniger interessante Zielprozessoren unterstützt werden; darunter Motorola 68k und Infineon C167 [62]. Letzterer war immerhin in der Vergangenheit in Motorsteuergeräten im Einsatz.

3.3 Hybride Prüfmethoden zur Laufzeitbestimmung

RapiTime von Rapita Systems Ltd.

RapiTime von Rapita Systems Ltd. aus York ist ein kommerzielles Laufzeitanalysetool. Es handelt sich genauso genommen um eine kommerzielle Version des Tools pWCET, das von der Real-Time-Systems Research Group an der University of York entwickelt wurde. Es verfolgt den Ansatz „das beste Modell für einen Prozessor, ist der Prozessor selbst".

RapiTime erhält die zu untersuchende Task in Form von Quellcode in C oder ADA und kann dann auf zwei verschiedene Arten Laufzeitinformationen gewinnen: Aktiv, durch eine automatische Instrumentierung des Quellcodes, oder Passiv, mittels hardwarebasierter Tracingmechanismen (ohne den Quellcode zu verändern) [73]. Auch ein CPU-Simulator kann zur Ablaufverfolgung eingesetzt werden [62]. RapiTime analysiert den Quellcode, partitioniert ihn in Grundblöcke und ermittelt eine Baumstruktur der Software, die auch zur Berechnung der Laufzeit eingesetzt wird (s. Abb. 3.4 (d)). Schleifenbegrenzungen werden aus Messungen oder durch Kommentare im Quellcode festgelegt. Ebenso kann die Tiefe der Instrumentierung und damit der Analyse im entsprechenden Modus durch Kommentare festgelegt werden [62]. Eine Kontrollflussanalyse ist durch disassemblieren auch am Objektcode möglich [73].

Als Ausgabe erhält der Benutzer eine detaillierte Auswertung des Laufzeitverhaltens einer Task. Es wird einerseits die maximale beobachtete Laufzeit berechnet und andererseits anhand einer Wahrscheinlichkeitsverteilung zusätzlich eine erwartete WCET bestimmt. RapiTime visualisiert die Testabdeckung, die mit den Eingangsdaten erzielt wurde, indem es den Quellcode farblich codiert darstellt. Auch in der dargestellten Baumstruktur wird die Testabdeckung farbig dargestellt. Zusätzlich wird ebenfalls sowohl im Quellcode als auch in der Baumstruktur farbcodiert angezeigt, ob der Code zur ermittelten Gesamt-WCET beiträgt. In zusammenfassenden Tabellen zeigt RapiTime die beobachtete und die berechnete WCET, die WCET der einzelnen Funktionen und deren Beitrag zum Gesamtergebnis sowie den Beitrag einzelner Aufrufkontexte der Funktionen [73].

Bewertung: Der große Vorteil an RapiTime ist seine Ausrichtung auf den industriellen Einsatz. Es ist durch seine kommerzielle Orientierung verhältnismäßig einfach zu bedienen (im Vergleich mit den Forschungstools) und erfordert nicht zwingend eine Kommentierung / Annotation des Quellcodes zur Laufzeitberechnung. Auch seine Integration in bestehende Prozesse ist einfach gehalten. Es gibt mehrere Möglichkeiten die Laufzeitdaten aus dem Zielsystem zu gewinnen, die teilweise unabhängig von der eingesetzten Hardwareplattform angewendet werden können. Im Prinzip muss für eine Anpassung an eine neue Plattform nur eine Möglichkeit geschaffen werden, die entsprechenden Daten aus dem Zielsystem zu gewinnen. RapiTime kann schon auf Modultestebene eingesetzt werden (statt im Integrationstest), was Kosten spart. Nachteilig an RapiTime ist vor allem die einem Messverfahren innewohnende Unsicherheit bezüglich der oftmals unerreichbaren vollständigen Pfadabdeckung oder Zustandsabdeckung. RapiTime versucht dies über Wahrscheinlichkeitsrechnung zu kompensieren und ermöglicht durch seine detaillierte Auswertung einen iterativen Testablauf und eine Variation der Eingangsdaten, um entsprechend höhere oder andere Testabdeckung zu erhalten. Außerdem ermöglicht es eine gezielte Optimierung der „WCET-Hotspots" (Bezeichnung nach [73]), indem es gezielt Codeteile markiert, die einen großen Anteil an der ermittelten WCET haben. Problematisch ist auch, dass zwar ein möglicher Worst-Case-Pfad ermittelt und darstellt werden kann, jedoch nicht die zugehörigen Eingangsdaten angeben werden können, da dieser aus einer Berechnung oder Abschätzung hervor geht.

SymTA/P

Das hybride WCET Tool SymTA/P (Symbolic Timing Analysis for Processes) wurde an der TU Braunschweig entwickelt. Es kann obere und untere Laufzeitgrenzen für in C geschriebene Programme bestimmen. SymTA/P vereint eine plattformunabhängige Softwareanalyse auf Quellcode-Ebene mit einer Messung auf der entsprechenden Plattform (auf Objektcodeebene). Diese Plattform kann ein Evaluation Board oder auch ein zyklenakkurater Simulator sein. Die Besonderheit an SymTA/P ist, dass es Grundböcke, die auf lediglich einem Pfad liegen, zu SFPs (Single Feasible Paths) [2, 3] zusammenfasst. Der Sinn dieser Zusammenfassung ist in erster Linie, dass die für die Messung erforderlichen Instrumentierungspunkte verringert werden können und damit bei der folgenden Messung weniger Overhead entsteht [62].

Da die Messung nicht mit einem kontrollierten und damit sicheren initialen Zustand der CPU ausgeführt werden kann, wird zur Sicherheit etwas Zeit hinzugerechnet. Bei der Ausführung muss die Task manuell so stimuliert werden, dass eine vollständige Zweigabdeckung erreicht wird. Allerdings werden dadurch nicht alle Pipelinezustände berücksichtigt, die an einem Knoten erreicht sein können. Auch hier wird ein Overhead addiert. Um das Cacheverhalten der Task zu analysieren, muss eine ununterbrochene Ausführung der Task sichergestellt werden. Außerdem muss ein Trace einer Taskausführung gewonnen werden, bei der garantiert keine Cache Misses auftreten. Speicherzugriffszeiten werden von SymTA/P auf der Zielhardware gemessen [62]. Die abschließende Berechnung erfolgt bei SymTA/P mittels IPET.

Bewertung: Vorteilhaft an SymTA/P ist der Ansatz, ein statisches Verfahren mit einem Messverfahren zu kombinieren. Auch sind bereits Messungen auf der TriCore Plattform mit diesem Tool durchgeführt worden.

Nachteilig sind bei SymTA/P sicherlich die vielen möglichen Fehlerquellen bei der Anwendung, vor allem im Bereich der Messung. Der Benutzer ist stark eingebunden in den Analyseprozess und das Ergebnis hängt teilweise stärker als bei anderen Tools von seinem Fachwissen und seiner Arbeitsqualität ab. Davon abgesehen wird an mehreren Stellen während der Analyse Overhead addiert, um unbekannte Zustände abzudecken. SymTA/P ist ein internes Tool der TU Braunschweig.

Hybrider Prototyp der TU Wien

Der hybride Prototyp der TU Wien vereint eine statische Programmanalyse mit einer Messung auf der Zielplattform um die Laufzeit zu ermitteln. Das Tool segmentiert und instrumentiert das eingegebene C-Programm und generiert automatisch Testdaten zur Stimulierung der Pfade in jedem Segment, deren Ausführungszeiten dann auf der Zielplattform gemessen werden. Zur Generierung von Testdaten geht das Programm in drei Schritten vor. Zuerst werden zufällige Eingangsdaten benutzt und die erreichte Pfadabdeckung wird mithilfe der Instrumentierung überprüft. Heuristische Methoden wie beispielsweise genetische Algorithmen verbessern die Pfadabdeckung weiter. Zum Schluss werden verbleibende Testdaten mittels Model Checking generiert. Dabei werden automatisch (falsche) unmögliche Pfade ausgegeben. Basierend auf den Pfadinformationen sowie den gemessenen Laufzeiten wird schließlich per IPET eine Obergrenze der WCET berechnet [62].

Das Tool stellt eine vollständige Pfadabdeckung innerhalb der Programmsegmente sicher. Prozessorzustände oder Ausführungskontexte werden dagegen nicht beachtet. Aus diesem Grund kann das Tool keine WCET unterhalb der ermittelten Grenze garantieren, wenn der Benutzer Gebrauch von modernen CPUs oder beschleunigenden Architekturfeatures macht.

Bewertung: Auch dieses Tool ist ein Forschungstool für den internen Gebrauch an der TU Wien und nicht öffentlich zugänglich. Die TriCore Architektur wird nicht explizit unterstützt, könnte aber relativ einfach in das Verfahren integriert werden. Als sinnvoll ist dies, ob der fehlenden Betrachtung von Cache und Pipeline, aber eher nicht anzusehen, da keine Laufzeitgarantie gegeben werden kann.

4 Temporales Prüfkonzept

Gezeigt wurde in den vorangegangenen Kapiteln die Notwendigkeit, bei der Entwicklung von Software für harte Echtzeitsysteme neben den funktionalen Eigenschaften auch die nicht funktionalen Eigenschaften, wie die obere Laufzeitgrenze, abzusichern. Neue Herausforderungen entstehen darüber hinaus bei der verteilten Entwicklung von Software. Hier ist es erforderlich, eine gemeinsame Komponentenbeschreibung zu definieren, damit Aussagen der fremdentwickelten Software-Komponenten mit den Ergebnissen der Systemkomponenten zusammengefasst werden können, um so das gesamte System abzusichern. Hierzu zählt die Beschreibung der oberen Laufzeitgrenze bzw. der maximalen Laufzeit der fremdentwickelten Software-Komponenten als Testfall, um eine Nachvollziehbarkeit bei allen Entwicklungspartnern zu ermöglichen.

Es wurden bestehende Ansätze bzw. Analyseverfahren zur Bestimmung von oberer und unterer Laufzeitgrenze sowie die hierbei auftretenden Probleme, wie Datenabhängigkeit und die architekturbedingte nicht konstante Programmpfadlaufzeit, beschrieben. Hierfür wurden Lösungsansätze auf der Basis von dynamischen, statischen und hybriden Prüfmethoden diskutiert.

Dynamische Prüfmethoden zur Laufzeitbestimmung, wie sie in Abschn. 3.1 diskutiert wurden, weisen dabei zusammengefasst die folgenden Probleme auf:

1. einen sehr hohen Aufwand für die Versuchsplanung,
2. die Schwierigkeit, sowohl Ziel- als auch Einflussgrößen festzulegen und diese in der Untersuchung zu messen, also quantitativ zu erfassen,
3. einen sehr hohen Aufwand für die Versuchsauswertung und
4. die Vollständigkeit der Ergebnisse.

Statische Prüfmethoden zur Laufzeitbestimmung haben im Wesentlichen den Nachteil, dass weder spezifizierte Testfälle auf ihr Laufzeitverhalten untersucht werden können, noch kann für die berechnete obere Laufzeitgrenze ein Testfall ausgegeben werden (s. Abschn. 3.2).

Hybride Prüfmethoden zur Laufzeitbestimmung zeigen ähnliche Schwächen wie statische und dynamische Prüfmethoden. Sie können jedoch spezifizierte Testfälle untersuchen, aber nicht für die berechnete obere Laufzeitgrenze einen Testfall ausgeben (s. Abschn. 3.3).

Das hier vorgestellte temporale Prüfkonzept stellt einen neuen Ansatz zur Lösung dieser Probleme bereit. Das Konzept gliedert sich in vier logische Schritte, die in zwei Phasen der Datengewinnung und der Datenanalyse zusammengefasst werden. Jeder logische Schritt ist eine abgeschlossene Folge von Tätigkeiten, die ein Zwischenergebnis liefern, das in einem anderen Schritt weiterverwendet wird. Die Schritte sind:

1. **Datengewinnung – Programmpfadbestimmung:** Die Bestimmung der ausführbaren Programmpfade erfolgt automatisch mit der Ausführung eines Testfalls. Diese Methode verzichtet auf eine aufwendige bzw. unvollständige Softwareanalyse (s. Abschn. 3.2.1), indem Synergien zu bestehenden funktionalen Prüfmethoden genutzt werden.
2. **Datengewinnung – Laufzeitbestimmung:** Die Laufzeitbestimmung erfolgt automatisch mit der Ausführung des Programms durch einen Testfall. Diese Methode berücksichtigt die Laufzeitabhängigkeit von Programmpfaden zu den Architekturmerkmalen, indem das Zielsystem als Ausführungsumgebung benutzt wird. Damit wird zugunsten einer hohen

Genauigkeit auf eine Nachbildung der Zielarchitektur verzichtet. Dies bedeutet auch, dass eine Betrachtung der entstehenden Schwierigkeiten, sowohl Ziel- als auch Einflussgrößen festzulegen und diese in der Untersuchung zu messen, also quantitativ zu erfassen, vorgenommen werden muss.

3. **Datenanalyse – Datenfusion:** Durch Datenfusion werden Programmpfadinformationen und Laufzeitinformationen über ihre Datenabhängigkeit kombiniert. Diese Methode ermöglicht es, Aussagen über pfadabhängige Laufzeiten bereitzustellen und damit Auswirkungen von Architekturmerkmalen auf die Programmpfadlaufzeit zu beurteilen.
4. **Datenanalyse – Struktursuche:** Bei der Struktursuche wird der längste Programmpfad bestimmt, um eine Prüfung der Laufzeit zu ermöglichen.

Weitere verfolgte Ziele sind, das temporale Verhalten der Software frühzeitig abzusichern und die für die Behandlung der nichtfunktionalen Eigenschaft - Rechenzeitbeschränkung - erforderlichen Techniken in den bestehenden Entwicklungsprozess zu integrieren. Für den Testfall zur Beschreibung der maximalen Laufzeit soll der geforderte Qualitätsnachweis auf Basis verlässlicher und abgesicherter Ergebnisse erbracht werden.

4.1 Prinzip

Dynamische Prüfmethoden, zu denen der hier beschriebene Ansatz zählt, basieren auf der Ausführung des Programms auf einem Laufzeitsystem und der Messung der Laufzeit. Wird die Ausführung des Programms mit definierten Eingaben durchgeführt, wird dies als Test bezeichnet. Wird gleichzeitig mit dem Test die Programmlaufzeit gemessen, so wird das hier als temporaler Test bezeichnet.

Definition 4.1 temporaler Test: Unter einem temporalen Test wird hier die Aufgabe des gezielten und systematischen Aufdeckens von Überschreitungen der maximal zulässigen Prozesslaufzeit $\Delta t_{\mathrm{Pr}\,A\max}$, die auf eine Verletzung der Rechenzeitbeschränkung hinweisen, verstanden.

Der temporale Test von Software ist die kontrollierte Ausführung eines Messobjektes[1], die der Überprüfung des temporalen Verhaltens des Programms dient. Der Test erfolgt unter definierten Bedingungen, bei dem die beobachtete und aufgezeichnete Ist-Laufzeit $\Delta t^{j}_{\mathrm{Pr}\,E,i}$ mit der maximal zulässigen Prozesslaufzeit $\Delta t_{\mathrm{Pr}\,A\max,j}$ (dem spezifizierten Prozess-Container) des Programms verglichen wird. Die abschließende Bewertung dient der Prüfung, ob das Programm die nach der Spezifikation geforderten Laufzeiteigenschaften erfüllt, d. h. die Bewertung, ob eine korrekte Umsetzung der temporalen Anforderungen erfolgte.

Der Test des temporalen Verhaltens der Software, wie er in dieser Arbeit beschrieben wird, gliedert sich aufgrund der ermittelten Laufzeiten in den Leistungstest und aufgrund der ermittelten Pfade in den White-Box-Test. Somit kann der hier beschriebene temporale Test dem Software-Modultest im Software-Entwicklungsprozess zugeordnet werden. Die vier logischen Schritte des temporalen Prüfkonzeptes werden im Folgenden problemorientiert als Teststrategie, temporaler Regressionstest, Messprozess, Datenanalyse und statische Programmcodeanalyse beschrieben.

Teststrategie: Die Formulierung der Strategie zum temporalen, dynamischen, strukturorientierten Test beinhaltet das Definieren von Anforderungen an den funktionalen Software-Modultest, die Definition von Anforderungen an die Testauswertung und die Definition von Anforderungen an den temporalen Test.

[1] Am Messobjekt (Testobjekt) werden nichtfunktionale (funktions- und strukturorientierte) Eigenschaften gemessen (beobachtet).

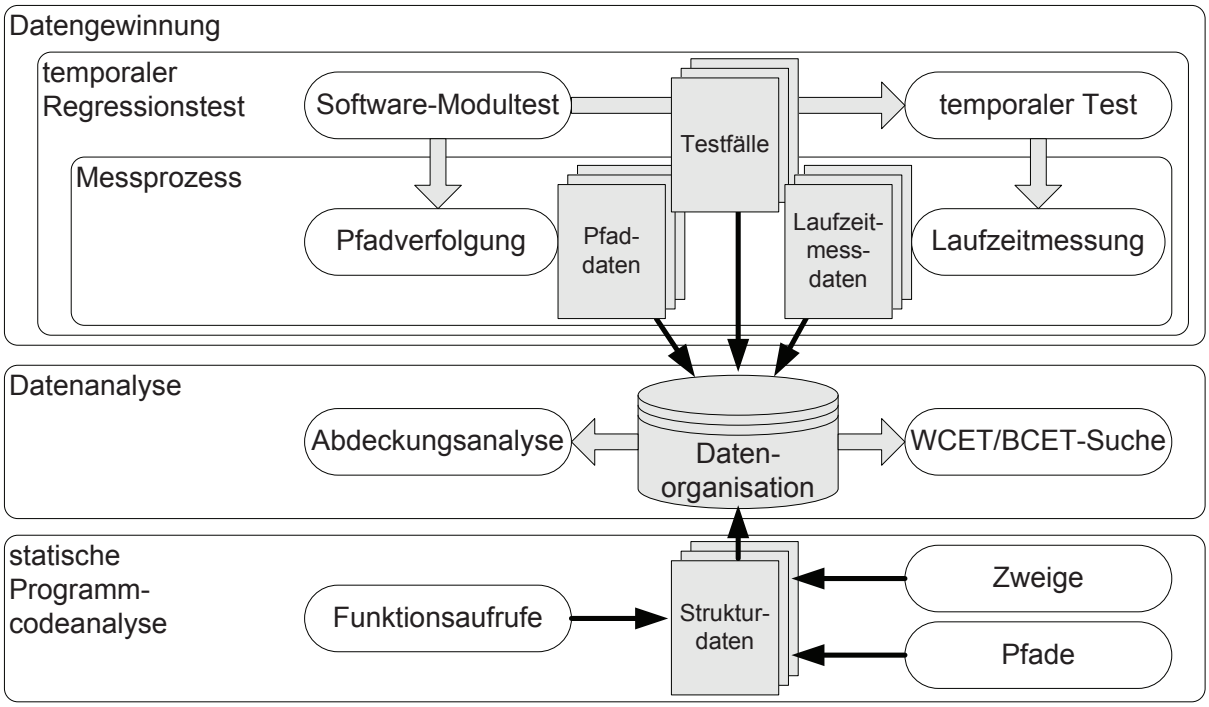


Abb. 4.1: Prüfkonzept zur pfadabhängigen Laufzeitbestimmung

Temporaler Regressionstest und Messprozess: Gezeigt wird ein Lösungsansatz zur Reduktion der Testaufwendungen als temporaler Regressionstest. Hierzu wird ein Messprozess definiert, der zur Erfassung der Zielgrößen (Prozesslaufzeiten) geeignet ist und es wird gezeigt, wie Einflussgrößen (Eingaben) miterfasst werden können. Zudem werden Zusammenhänge zwischen Einflussgrößen und Zielgrößen über die Pfadabhängigkeit erfasst. Hierzu wird ein Vorgehen beim Messen festgelegt.

Datenanalyse: Mit dem Konzept der Datenanalyse soll vor allem den hohen Aufwendungen für eine Versuchsauswertung begegnet werden. Daher wird im Konzept die Definition der Inhalte der Daten vorgegeben und das Modell zur Datenorganisation vorgestellt. Die darauf aufbauende Zuordnung der Daten sowie die statische Auswertung und Bewertung der Daten werden ebenfalls dargestellt.

Statische Programmcodeanalyse: In einem weiteren Schritt wird eine Ergänzung des Prüfkonzeptes vorgeschlagen, um ein vollständiges Ergebnis zu erhalten und bestehende repräsentative Testfälle besser als bisher beurteilen zu können. Hierfür werden Anforderungen an eine statische Programmcodeanalyse, eine Erweiterung der temporalen Teststrategie, eine Modifikation des Messverfahrens und der Aufbau des Datenanalysemodells definiert.

4.2 Strategie zum temporalen Test

Die Strategie des temporalen Tests umfasst das Beobachten und Aufzeichnen der Ist-Laufzeit und das Vergleichen mit der maximal zulässigen Laufzeit. Die Vollständigkeit des temporalen Tests wird durch die Pfadabdeckung des Programms unter definierten temporalen Bedingungen[2] bestimmt.

[2] Die temporalen Bedingungen werden durch die laufzeitbeeinflussenden Faktoren der Echtzeitumgebung bestimmt. Eine Einbeziehung aller laufzeitbeeinflussenden Faktoren ist ein erschöpfender temporaler Test, dieser Test bezeichnet die Prüfung einer Software mit allen möglichen Eingaben in allen denkbaren Betriebssituationen des Echtzeitsystems. Der erschöpfende Test besitzt keine praktische Bedeutung [28].

Zu den Problemen des temporalen Tests gehören:

- das Risiko von ungetesteten Programmpfaden und
- der hohe Aufwand für die Testfallerstellung.

Es existieren verschiedene Ansätze zur Testauswahl [13], die sich im Wesentlichen auf Benutzervorgaben für Testeingaben und zufällige Testeingaben reduzieren lassen. In beiden Fällen ist bei komplexen Programmen und Hardwarearchitekturen nicht sichergestellt, dass die maximale Laufzeit bzw. die obere Laufzeitgrenze tatsächlich ermittelt wird.

An dieser Stelle soll kein neues Konzept zur Testfallerstellung, wie z. B. in [62], präsentiert werden, sondern bereits bestehende dynamische, funktions- und strukturorientierte Prüftechniken (s. Abschn. 2.3.2) auf ihre Anwendbarkeit für den temporalen Test bewertet werden. Keine der bisherigen dynamischen Prüfmethoden zur Laufzeitbestimmung versuchte hier Synergien zur Testfallerstellung aus dem funktions- und strukturorientierten Test zu nutzen. Dabei sind die Anforderungen an den Software-Modultest und den temporalen Test sehr ähnlich. In der Tabelle 4.1 ist eine Auswahl von Gemeinsamkeiten dargestellt. Eine Betrachtung der Praxis zeigt, dass gerade dem Software-Modultest ein hoher Stellenwert bei der Absicherung von programmierbaren elektronischen Systemen eingeräumt wird. Dies spiegelt sich in Standards und Normen wieder, wie z. B. bei der DIN61508 [81] oder ISO/WD26262, in denen Testziele in Abhängigkeit der funktionalen Sicherheit vorgeben werden.

Wie die Zusammenstellung (Tabelle 4.1) zeigt, ist das Problem der Testfallerstellung für den temporalen Test auch bei der Testfallerstellung für den Software-Modultest zu finden.

Tabelle 4.1: Merkmale von Software-Modultest und temporalen Test

Merkmale der Testsituation	Software-Modultest	temporaler Test
Testgrundlage	- Kontrollflussgraph - Spezifikation	- Kontrollflussgraph
Testauswahl	- Datenabhängigkeitsanalyse[a] - Spezifikationsanalyse	- Datenabhängigkeitsanalyse[a]
Testvollständigkeit	- Strukturabdeckung - Spezifikationsabdeckung	- Pfadabdeckung

a - Analyse von (Eingaben) Daten, die den Programmpfad beeinflussen

Neuer Ansatz: Durch geschickte Ausnutzung bestehender funktions- und strukturorientierter Tests können auf relative einfache Weise Testfälle für den temporalen Test bereitgestellt werden, ohne den Aufwand des Software-Modultests wesentlich zu erhöhen.

Der Beitrag des folgenden Abschnittes liegt in der Aufwandsbegrenzung für die Versuchsplanung (einschließlich der Testfallerstellung) zum temporalen Test. Hierfür werden die vorgegebenen Minimalkriterien der Software-Prüfung im gewöhnlichen Kontext, wie sie in Kapitel 2 vorgestellt wurden, für eine temporale Absicherung erweitert. Der funktions- und strukturorientierte Test muss die hier beschriebenen erweiterten Kriterien erfüllen, um sinnvolle Testfälle für den temporalen Test zu liefern. In den folgenden Abschnitten wird die Datengewinnung zum temporalen Test beschrieben. Am Schluss dieses Kapitels werden die Ergebnisse aus Programmpfad- und Laufzeitbestimmung dargestellt.

4.2.1 Anforderungen an den funktionalen Test

Die Ergänzung des funktionalen Tests [84] dient der Beurteilung von temporalen und funktionalen Zusammenhängen. Weiterhin wird in allen Testphasen eine funktionsorientierte Test-

durchführung verfolgt. Ergänzt werden muss diese durch eine Überdeckungsanalyse im Software-Modultest zum Nachweis der erreichten Testabdeckung als Metrik für die qualifiziert geleistete Testaktivität.

Anforderungen an den Software-Codetest

Annahme: Generell wird davon ausgegangen, dass die Korrektheit der Funktion im Funktionstest und die Softwarequalität im Software-Modultest nachgewiesen wurden, bevor der temporale Test durchgeführt wird.

Begrenzung der Software-Komplexität: Für den temporalen wie auch für den funktions- und strukturorientierten Test müssen Programmierkonventionen geprüft werden. Das heißt Software-Metriken [82] müssen erstellt werden, die z. B. die Anzahl der Programmpfade und Zweige feststellen. Die Modellierungs- und Codierungsrichtlinien müssen hierfür Vorgaben bereithalten, die das Testobjekt (Programm) auf ein testbares Maß begrenzen. Vorgaben bestehen hierfür nur für manuell erstellten C-Code, z. B. das vom Arbeitskreis „Software Test" der HIS definierte „Gemeinsame Subset der MISRA[3] C Guidelines" [8]. In dieser Arbeit werden diese Regeln auch für generierten Code verwendet. Es muss jedoch genau betrachtet werden, welche Regeln wirklich aussagekräftig sind und welche - da sie durch generierten Code erst gar nicht verletzt werden können - nicht zur objektiven Steigerung der Qualität der Software beitragen. Die HIS hat eine Grundmenge an Metriken festgelegt, welche abgeprüft werden sollten. In Anlehnung daran hat die Qualitätssicherung die eigens relevanten Metriken bestimmt und in einem Qualitätsmodell zusammengefasst. Die Tabelle 4.2 enthält einen Ausschnitt des Qualitätsmodells.

Anmerkung: In dieser Arbeit erfolgt keine kritische Betrachtung der MISRA-Regeln in Bezug auf generierten C-Code. Dies erfolgt bereits in diversen Arbeitskreisen. Abzusehen ist jedoch, dass die Regeln für Autocode angepasst und erweitert werden müssen.

Tabelle 4.2: Software-Metriken aus dem Qualitätsmodell

Erklärung der Metrik	Metrikkürzel	Grenzen laut Qualitäts-modell
Zyklomatische Komplexität	VG	≤ 20
Durchschnittliche Größe eines Statements einer Funktion	AVGS	≤ 9
Anzahl der ausführbaren Statements einer Funktion	STMT	≤ 50
Anzahl der GOTO-Anweisungen	GOTO	0
Anzahl der Austrittspunkte aus einer Funktion	RETU	1
Anzahl der aufrufenden Funktionen einer Funktion	NBCALLING	≤ 10
Anzahl der Ausführungspfade der Funktion	PATH	≤ 1000
Anzahl der Übergabeparameter einer Funktion	PARA	≤ 5
Anzahl von Rekursionen	Kein Kürzel	0
Anzahl Statements, die nicht benutzt werden; „Toter Code"	NBCALLING	0
Anzahl aufgerufener Funktionen einer Funktion	DC_CALLING	≤ 7
Maximale Schachtelungstiefe der Kontrollstruktur einer Funktion	LEVL	≤ 4

Die zyklomatische Komplexität oder auch McCabe-Metrik ist ein Indikator für Fehleranfälligkeit und Pflegbarkeit. Sie untersucht die Komplexität der Struktur und bezieht dabei gezielt Verzweigungen mit ein. Sie setzt sich zusammen aus der Anzahl der Knoten (n), der Anzahl

[3] Motor Industry Software Reliability Association

der Kanten (e) und der Anzahl der Programmelemente/-komponenten (p). Die Berechnung erfolgt nach der Formel V(G) = e – n + 2p. Allerdings sagt die V(G) nichts über die Struktur des Programms aus.

Die durchschnittliche Größe eines Statements einer Funktion setzt sich zusammen aus: AVGS = (N1 + N2 + 1) / (STMT + 1). Es ist umso schwerer, eine Funktion zu verstehen, je größer diese ist. N1 gibt die Gesamtzahl der Operatoren und N2 die Gesamtzahl der Operanden an. STMT ist die Anzahl der ausführbaren Statements einer Funktion (vgl. Metrik 3), was sich maßgeblich auf die Verständlichkeit des Programms auswirkt. Diese Faktoren beeinflussen die Fehlersuche und die Änderbarkeit negativ.

GOTO-Anweisungen werden nicht zugelassen, denn sie reduzieren erheblich die Testbarkeit und erhöhen die Anzahl der Pfade. RETU gibt die Zahl der Austrittspunkte einer Funktion an. Es ist geduldet, dass die Funktion keinen Austrittspunkt hat, allerdings sollte nie mehr als einer vorhanden sein. Die Metrik NBCALLING wird in zweierlei Hinsicht interpretiert. Sie gibt nicht nur an, wie viele Funktionen von einer Funktion aufgerufen werden, sondern auch, ob die Funktionen mindestens einmal aufgerufen werden. Ist letzteres nicht der Fall, handelt es sich um „toten“ Code, d. h. Code der nicht benutzt wird und somit auch nicht zur Funktionalität beiträgt, aber dennoch Speicherplatz verbraucht. Außerdem ist „toter“ Code laut der IEC 61508 [81] nicht erlaubt.

Im Code dürfen keine Rekursionen vorhanden sein, da dies zu erhöhtem - evtl. nicht kontrollierbarem - Speicherverbrauch und der Erhöhung der Laufzeit bzw. Rechenzeit führt, so dass die Echtzeitfähigkeit nicht mehr garantiert werden kann. Die Anzahl der Ausführungspfade der Funktion (PATH) ist das wichtigste Kriterium für die Testbarkeit. Um eine hohe Testabdeckung zu erreichen, sollte die Pfadanzahl möglichst klein gehalten werden. Weiterhin darf, um die Übersichtlichkeit bzw. das Verständnis des Programmcodes und die Änderbarkeit positiv zu beeinflussen, eine bestimmte Verschachtelungstiefe (LEVL) nicht überschritten werden. Gleiches gilt für die Anzahl der aufgerufenen Funktionen (DC_CALLING). Auch die Anzahl der Parameter (PARA), welche einer Funktion übergeben werden, bestimmt in gewisser Weise die Komplexität der Funktion. Zudem wird hierdurch Einfluss auf den Stack-Bedarf genommen. Hinzu kommt die Prüfung von Restriktion für Schleifen im Programmcode, hierfür muss im Funktionsmodell explizit die Schleifengrenze festgelegt sein.

Anforderungen an den funktions- und strukturorientierten Software-Modultest

Notwendig ist eine funktionsorientierte Testplanung mit geeigneter, systematischer funktionsorientierter Testtechnik. Diese muss durch strukturorientierte Testtechniken unterstützt werden. Vor der Durchführung der funktionalen Testfälle ist das zu testende Modul unter Kontrolle eines Zweigüberdeckungswerkzeuges zu bringen, das ein Aufzeigen der erreichten Zweigüberdeckung ermöglicht. Hier müssen Werkzeuge bereitgestellt werden, um eine Verfolgung der durchlaufenen Programmpfade zu realisieren.

Definition 4.2 Testfall: Ein Testfall ist ein vollständiger Eingabedatensatz (Eingabewerte) d, zu dem ein Soll-Ergebnis a angegeben wird.

Wird der Testfall ausgeführt, durchläuft das Programm deterministisch einen Programmpfad p, $p \in P$ und zeigt damit das Ist-Verhalten mit dem Ist-Ergebnis a'. Zu jedem Testfall gehört die Angabe definierter Vorbedingungen, unter denen der Test zu erfolgen hat. Die funktional geplanten Testfälle müssen vollständig durchgeführt werden. Hierauf erfolgt die Kontrolle der Zweigüberdeckung. Anschließend werden die Ursachen für nicht ausgeführte Zweige ermittelt.

Definition 4.3 Testpfad: Ein Testpfad ist ein durch einen Testfall ausgeführter Programmpfad p, $p \in P$, der mit pm, $pm \in PM \subseteq P$ und $PM \subseteq AP$ bezeichnet wird.

Die Testdaten für nicht ausgeführte Zweige werden zusätzlich erzeugt. Das akzeptierte Minimum an Abdeckung stellt die vollständige Zweigabdeckung dar (s. [81]). Es sollte angestrebt werden, mit den funktionalen Testfällen diese Zweigüberdeckung zu erreichen. Strukturorientierte Tests kommen daher nur für Zweige, die nicht durch funktionsorientierte Tests erreicht wurden, infrage.

Eine Erweiterung dieser Anforderungen betrifft das Testen von Schleifen, hier sind die folgenden Testanforderungen umzusetzen:

1. der strukturierte Pfadtest und boundary interior-Pfadtest [85],
2. die modifizierte boundary interior-Testtechnik [54] und
3. es ist die maximale Anzahl von Schleifendurchläufen zu testen und der Versuch diese zu überschreiten [86].

Anforderung an die Testauswertung

Die Testauswertung unterscheidet sich nicht wesentlich von der bisher in der Praxis vollführten Testauswertung. Zum Soll-Ist-Vergleich der Testauswertung kommt eine Zuordnung des ausgeführten Programmpfades zum jeweiligen Testfall hinzu. Die verifizierten Testergebnisse des funktionalen und strukturellen Tests müssen zusammen mit den Testeingaben für einen gewöhnlichen Regressionstest als Referenztestfall aufbereitet werden.

Definition 4.4 Referenztestfall: Ein Referenztestfall ist ein vollständiger Eingabedatensatz (Eingabewerte) d mit zugehörigem verifizierten Testergebnis (Ausgabewerte) a und dem ermittelten Testpfad pm, $pm \in PM$.

Der anschließende temporale Test erfolgt auf Basis der Testeingaben aus dem Software-Modultest. Die Zusammenhänge zwischen Software-Modultest und temporalen Test werden im Abschn. 4.3 beschrieben.

4.2.2 Anforderungen an den temporalen Test

Vor der Durchführung des temporalen Tests ist das zu testende Programm unter Kontrolle eines Zeitmesswerkzeuges zu bringen. Das Zeitmesswerkzeug gibt für jeden Testfall die Laufzeit des jeweiligen Testpfades zurück.

Definition 4.5 temporaler Testfall: Ein temporaler Testfall ist ein vollständiger Eingabedatensatz (Eingabewerte) d, zu dem eine maximal zulässige Prozesslaufzeit $\Delta t_{\mathrm{Pr}\,A\max}$ angegeben wird.

Die Testfallerstellung, d. h. die Vorgabe der Eingabewerte, kann durch Benutzervorgaben oder durch zufällige Testeingaben erfolgen. Für einen temporalen Testfall muss kein Testergebnis (Soll-Ausgabewert) angegeben werden, dies ist durch den funktions- und strukturorientierten Test erreicht.

Definition 4.6 ausgeführter temporaler Testfall: Ein ausgeführter temporaler Testfall besteht aus einem vollständigen Eingabedatensatz (Eingabewerte) d mit dem zugehörigen Testergebnis (Ist-Ergebnis) a'' und einer Prozesslaufzeit $\Delta t_{\mathrm{Pr}\,E,i}$, $\Delta t_{\mathrm{Pr}\,E,i} \in \Delta T_{\mathrm{Pr},i}$. Dabei ist $\Delta T_{\mathrm{Pr},i}$ die Menge aller Laufzeiten des Prozesses i.

Temporale Beeinflussung: Die Prozesslaufzeit wird hier auf ein konkretes Programm bezogen, das als Prozess auf einer bestimmten Hardware ausgeführt wird. Mit Prozesslaufzeit ist die Zeit gemeint, die das Programm für die Bearbeitung einer bestimmten Eingabe - dem temporalen Testfall - unter wiederholbaren Bedingungen benötigt. Wiederholbare Bedingungen bedeuten, dass bei wiederholter Messung derselben Messgröße die beherrschbaren Bedingungen ungeändert bleiben. Für die so gewonnenen Messwerte bleibt die systematische

Messabweichung e_s gleich. Den Idealfall stellen Wiederholbedingungen dar, bei denen all diejenigen Bedingungen ungeändert bleiben, deren Änderungen als Ursache für die Änderung systematischer Messabweichungen infrage kommen. Beherrschbare und unbeherrschbare Bedingungen sind z. B. Messverfahren, Messeinrichtung, spezielle Einflussgrößen (Testfall), Messobjekt, Architekturmerkmale (Cachezustände, Pipelinebelegung, usw.), Beeinflussung durch Multitasking-Umgebung und Ressourcenkonflikte. Die Kenntnis von diesen Bedingungen soll im Folgenden dafür benutzt werden, um Messabweichung, Korrektur und Messunsicherheit anzugeben.

Funktionale Beeinflussung: Abhängig vom Messverfahren wird durch die Messung das Messobjekt nachhaltig verändert. Messverfahren, die auf einer Instrumentierung des Programmcodes aufbauen, wie sie in Abschn. 3.1 beschrieben wurden, verändern das Programm und damit das Messobjekt. Die mit dem Test durchzuführende Laufzeitmessung hat damit eine Rückwirkung auf das Programm. Durch ein Zeitmesswerkzeug können somit Fehler im Programm bzw. Programmablauf entstehen. Bisherige Verfahren gehen auf dieses Problem nicht ein. Es wird darauf vertraut, dass:

1. die Instrumentierung ohne funktionalen Einfluss bleibt und
2. dass die Instrumentierungspunkte immer an der richtigen Stelle im Programmfluss platziert werden.

In dieser Arbeit werden daher zwei Ansätze verfolgt. Im ersten Ansatz wird die Instrumentierung zusammen mit der Codegenerierung vorgenommen und damit nachträgliche Änderungen am einmal getestetem Programm umgangen. Der Ansatz wird im Abschn. 4.4 mit dem Prinzip der Datengewinnung beschrieben. Als zweite Neuerung wird hier ein Verfahren vorgestellt, das eine Erkennung von Seiteneffekten durch das Zeitmesswerkzeug ermöglicht. Dabei erfolgt die Testfallvorgabe zum temporalen Test auf Basis von Benutzervorgaben. Der Benutzer greift dafür auf die Referenztestfälle zurück. Das Verfahren wird im folgenden Abschnitt als temporaler Regressionstest beschrieben.

4.3 Temporaler Regressionstest

Durch die Übernahme der Testfälle aus dem Software-Modultest wird der temporale Test zu einem temporalen Regressionstest. Der temporale Regressionstest (s. Abb. 4.2) ist eine Wiederholung von bereits durchgeführten dynamischen, funktions- und strukturorientierten Testabläufen. Nachdem die Korrektheit der erzeugten Testergebnisse der Softwareversion n' anhand der Spezifikation verifiziert worden ist, bezeichnet man die aufgezeichneten Testeingaben und Testergebnisse als Referenztestfälle. Die zu testende Softwareversion n'' ist pro durchzuführendem Testfall in den gleichen Zustand zu versetzen wie bei der vorhergehenden Durchführung (mit n'), d. h., mit identischen Eingaben (Referenzdaten) zu beschalten. Die erzeugten Ausgaben der Softwareversion n'' (Laufzeitkontrolle) sind mit den Ausgaben der Softwareversion n' (Zweigkontrolle) zu vergleichen. Werden keine Unterschiede festgestellt $(a' = a'')$ bei $(d' = d'')$, war der temporale Regressionstestfall erfolgreich, und es kann eine Zuordnung von Testfall n, Eingabe d_n, Ergebnis a_n, Programmpfad pm_n und Laufzeit $\Delta t_{\mathrm{Pr}E}$ erfolgen. Bei Unterschieden ist zu prüfen, wodurch dieses Fehlverhalten verursacht wird, d. h., bei gleicher Eingabe unterschiedliche Ergebnisse $(a' \neq a'')$ erzeugt werden. Der Fehler kann aus den beschriebenen Seiteneffekten der Instrumentierung oder durch fehlende Eingangs- oder Zustandsgrößen bei der Synchronisierung verursacht sein (unvollständiger Eingabedatensatz d). In jedem Fall muss der Fehler lokalisiert und korrigiert werden. Die Testfälle müssen wiederholt werden und alle Eingangs- und Zustandsgrößen müssen zur Synchronisation gemessen werden. Aus wirtschaftlichen Überlegungen sollte für die wiederholte Abarbeitung stets gleichbleibender Schritte ein Werkzeug verwendet werden.

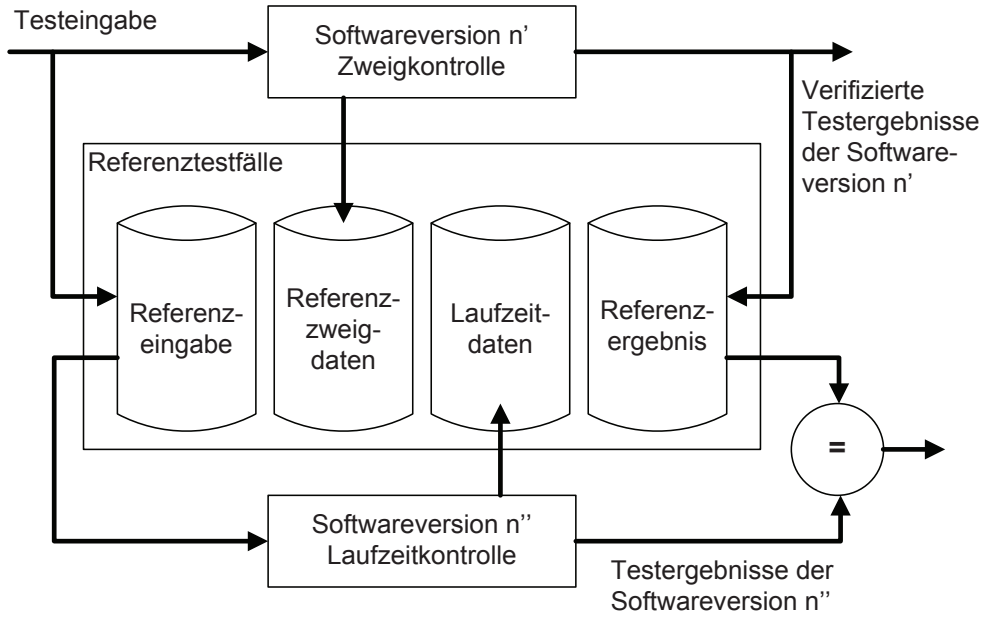


Abb. 4.2: Aufbau des temporalen Regressionstests

Statt der Synchronisierung über die Eingabedaten kann auch eine Synchronisierung über die Testfallnummern[4] erfolgen, wenn sichergestellt werden kann:

- dass die zu testende Software pro durchzuführenden Testfall in den gleichen Zustand zu versetzen ist, wie bei der vorhergehenden Durchführung, und
- dass immer identische Eingaben vorgegeben werden.

Bei einer manuellen Durchführung des Tests ist die Wahrscheinlichkeit, dass hierbei Fehler auftreten, recht hoch, hier ist immer eine Synchronisierung über die Eingangsdaten vorzunehmen. Durch Automatisierung wird der Aufwand für manuelle Wiederholungen der Testfälle eingespart.

Die temporalen Regressionstestwerkzeuge, wie z. B. HiL-Systeme, müssen in der Lage sein, die Testeingaben und Testergebnisse sowie Laufzeit- und Pfadinformationen bei der Testdurchführung aufzuzeichnen. Im Allgemeinen zeichnen Regressionstestwerkzeuge lediglich die Aktivitäten an den Schnittstellen auf - Eingabe und Reaktion - unabhängig von der Programmiersprache der zu testenden Software. Die zusätzlichen Informationen über Programmpfad und Laufzeit müssen ebenfalls über diese Schnittstellen bereitgestellt werden. Mit dem Problem der Erfassung von Ziel- und Einflussgrößen wird sich im folgenden Abschnitt befasst.

4.4 Messprozess zur pfadabhängigen Laufzeitbestimmung

Der folgende Abschnitt setzt sich mit der quantitativen Erfassung von Ziel- und Einflussgrößen durch Messung und den damit verbundenen Problemen auseinander.

Beschreibung von Ziel- und Einflussgrößen: Das Problem der Eingabedatenabhängigkeit der Prozesslaufzeit beschreibt die Merkmale Eingabe und Laufzeit, die erfasst bzw. gemessen werden müssen. Ein weiteres Merkmal ist der Programmpfad, anhand dessen die Beurteilung des Tests erfolgt. Die Prozesslaufzeit $\Delta t_{\mathrm{Pr}\,E}$ als Zielgröße muss messbar gemacht werden, ebenso die zugehörige Eingabe d_n als Einflussgröße. Jede Messung besteht damit aus einem

[4] Fortlaufende Nummerierung der Testeingaben

Merkmalspaar. Das erste Merkmal ist immer der vollständige Eingabedatensatz $d_{n,i}$ für den Prozess i (Pr_i). Mit diesem Datensatz $d_{n,i}$ durchläuft Pr_i deterministisch einen ganz bestimmten Programmpfad, den Pfad $p_{n,i}$. Das zweite Merkmal ist die Prozesslaufzeit $\Delta t^j_{\mathrm{Pr}\,E,i}$ von Pr_i. Eine Sortierung der Merkmalspaare $[d_{n,i}; \Delta t^j_{\mathrm{Pr}\,E,i}]$ erfolgt über dem Zeitpunkt der Messung bzw. über die j-te Ausführung des Programms als Prozess.

Eine weitere Abhängigkeit besteht zwischen der Eingabe $d_{n,i}$ für Pr_i und dem durchlaufenen Programmpfad $p_{n,i}$, der aus einer vollständigen Sequenz von Knoten $(n_{start}, n_1, ..., n_m, n_{stop})$ besteht. Der Programmpfad als drittes messbares Merkmal, zusammen mit der zugehörigen Eingabe, kann ebenfalls als Merkmalspaar $[d_{n,i}; p_{n,i}]$ angegeben werden. Damit kann der Bezug der Eingabedatenabhängigkeit der Prozesslaufzeit (mit dem Merkmalspaar $[d_{n,i}; \Delta t^j_{\mathrm{Pr}\,E,i}]$) und der Eingabedatenabhängigkeit des Programmpfades (mit dem Merkmalspaar $[d_{n,i}; p_{n,i}]$) über die Eingabe $d_{n,i}$ hergestellt werden.

Problemstellung: Bei der objektiven[5], reproduzierbaren[6] und quantitativen[7] Erfassung der beschriebenen Ziel- und Einflussgrößen müssen die Messgrößen nicht nur wertemäßig erfasst werden, sondern auch mit der Angabe der Genauigkeit belegt werden. Die hieraus resultierenden Fragen nach dem Messprinzip, dem Monitorsystem und der Testumgebung werden im folgenden Abschnitt beantwortet. Die Frage der Genauigkeit wird im Zusammenhang mit der Bewertung der Laufzeitmessergebnisse im Kapitel 6 beantwortet. Hierfür ist die Betrachtung der Implementierung des Messprinzips in ein Messverfahren und den dabei eingesetzten Messgeräten notwendig.

4.4.1 Messprinzip

Laufzeitmessung

Das hier beschriebene Messprinzip folgt im Wesentlichen dem in Abschn. 3.1.3 beschriebenen Ansatz der Abfrage eines Hardware-Timers. Hierbei wird für die Zeitmessung ein interner Zeitbaustein des Prozessors genutzt. Diese Zeitbausteine (Hardware-Timer) sind Zähler, die seit dem Start des Systems die Taktzyklen synchron zum Prozessortakt zählen. Die Laufzeitmessung erfolgt durch das Abfragen des Hardware-Timers vor und nach der Ausführung des zu messenden Programms (s. Abb. 4.3). Durch Bildung der Differenz aus den Timerwerten kann die Anzahl der Taktzyklen (Timerzyklen) zur Abarbeitung des Messobjektes (Programm) bestimmt werden. Die Taktzyklen lassen sich dann in Abhängigkeit von der Taktung und Synchronisierung zum Prozessor in eine Zeiteinheit umrechnen. Die Genauigkeit der Laufzeitbestimmung ist abhängig von der Auflösung des Hardware-Timers und dem Einfluss der Abfrage des Hardware-Timers auf das Messobjekt. Das Abfragen des Hardware-Timers erfolgt durch Instrumentierung, d. h. durch eine Erweiterung des Programmcodes. Hierfür werden Analysemarken eingefügt. Diese Analysemarken ermöglichen es, bei der Ausführung des Programms den Hardware-Timer auszulesen und auszugeben.

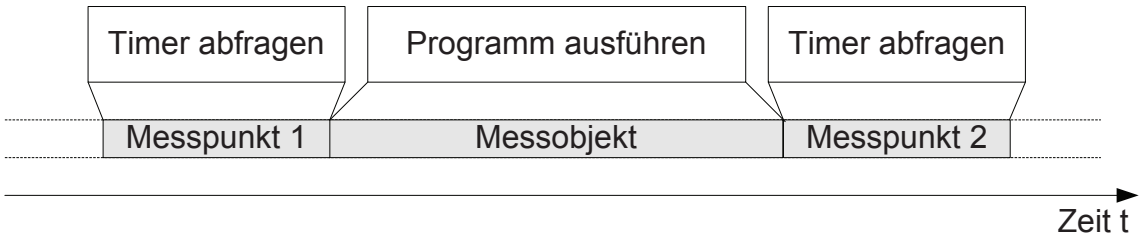


Abb. 4.3: Messprinzip zur Laufzeitmessung

[5] unabhängig vom Anwender
[6] wiederholbar und kontrollierbar
[7] mit einer Zahl versehen

Drei prinzipielle Arten von Analysemarken können unterschieden werden:

1. die Veränderung des Quelltextes durch zusätzliche Befehle,
2. die Veränderung des bereits kompilierten Programms durch das Einfügen von Befehlen und
3. die Instrumentierung des Programmaufrufes (das aufgerufene Programm bleibt unverändert).

Bei dem hier beschriebenen Ansatz zur Laufzeitmessung wird ein hoher Automatisierungsgrad angestrebt. Dies soll unter anderem durch automatisierte Instrumentierung mit der Codegenerierung erfolgen, d. h., der Quellcode wird in einem Schritt generiert und instrumentiert. Die Möglichkeit 2 kann daher nicht eingesetzt werden, da durch die Codegenerierung bereits vor dem Kompilieren instrumentiert wird. Die Möglichkeit 3 wird ausgeschlossen, da der Codegenerator außerhalb des generierten Programms instrumentieren müsste.

Fehlerbetrachtung Laufzeitmessung: Die Laufzeiteinflüsse von instrumentierenden Verfahren hängen im Wesentlichen von der Architektur des Zielsystems und von dem messtechnischen Verfahren ab, das die Signale der Instrumentierung aufnehmen kann.

Bei komplexen Hardwarearchitekturen ergibt sich neben der direkten Beeinflussung der Programmausführungszeit durch die Ausführung der Instrumentierung auch eine indirekte Beeinflussung durch eine Veränderung der Belegung von z. B. Registern, Cache und Pipeline. Ziel ist es diese Beeinflussung zu vermeiden oder die Messabweichung e vollständig zu bestimmen, damit diese korrigiert werden kann. Wie in Kapitel 3 bereits beschrieben, wird bei bestehenden Messverfahren zur Laufzeitbestimmung auf eine vollständige Beschreibung der Messabweichung verzichtet. Dies resultiert vor allem daher, dass durch Cache und Pipeline ein unbekannt bleibender Anteil $e_{s,u}$ der systematischen Messabweichung und die nicht genau feststellbare zufällige Messabweichung e_r, wie z. B. durch Ressourcenkonflikte, zu finden ist. Daher sollen möglichst wenige Analysemarken eingefügt werden. Der bisher betrachtete Ansatz sieht lediglich eine Instrumentierung am Anfang und am Ende des Prozesses zur Laufzeitmessung vor (s. Abb. 4.3). Die Gründe, die zu mehr als zwei Analysemarken je Prozess führen, werden im Kapitel 7 diskutiert.

Eindeutigkeit von Analysemarken: Müssen mehrere Prozesse gleichzeitig gemessen werden, bedeutet dies, dass mehr als zwei Analysemarken je Messung aufgezeichnet werden. Damit muss die Eindeutigkeit der Analysemarken sichergestellt werden. Die Eindeutigkeit kann durch:

- die Zuweisung von definierten Speicheradressen für die Aufnahme der Messwerte jeder Analysemarke oder
- durch die Vergabe einer eindeutigen Nummer je Analysemarke erfolgen.

Definierte Speicher/Register bedeuten eine Abhängigkeit vom Zielsystem. Die Implementierung von Analysemarken verlangt damit immer eine zielsystemabhängige Anpassung. Das Einfügen einer Nummer bedeutet zusätzliche Instruktionen und zusätzliche Daten, die je Analysemarke übertragen werden müssen. Die Ausführungszeit einer Analysemarke wird damit erhöht. Aufgrund der zielsystemunabhängigen Lösung wird nachfolgend diese Instrumentierung verfolgt. Für die weitere Betrachtung werden Analysemarken mit Nummer ohne weitere Differenzierung als Messpunkt mit $(t_1,...,t_m)$ bezeichnet.

Pfadmessung

Um die Abhängigkeit zwischen der Eingabe $d_{n,i}$ und dem durchlaufenen Programmpfad $p_{n,i}$ zu messen, wird der Programmcode an relevanten Stellen mit speziellen Instruktionen versehen, welche Aufschluss über die Abarbeitung des Programms geben. Je nach der Platzierung

der Instrumentierung kann dadurch eine Aussage über die Anweisungs-, Zweig-, Pfad-, Funktions- oder Messsegmentabdeckung getroffen werden.

- Eine Anweisung entspricht hier einem Knoten im Kontrollflussgraphen.
- Ein Zweig entspricht einer gerichteten Kante im Kontrollflussgraphen.
- Ein Pfad entspricht einer Sequenz von Knoten und Kanten, die beim Startknoten beginnt und mit dem Endknoten des Prozesses endet.

Im Gegensatz zum allgemeinen Anwendungsfall, der Bestimmung der Testüberdeckung [13], muss für den hier beschriebenen temporalen Test für jeden Pfad $p_{n,i}$ die vollständige Sequenz von Knoten $(n_{start}, n_1, ..., n_m, n_{stop})$ angegeben werden. Alternativ dazu kann die vollständige Sequenz von Kanten (Zweigen) $(z_1, ..., z_m)$ für $p_{n,i}$ angegeben werden (s. Abb. 4.4). Jeder Testfall n, bestehend aus der Eingabe $d_{n,i}$, muss eindeutig dem Programmpfad $p_{n,i}$ zugeordnet werden können.

Anmerkung: Anstelle der üblichen Bezeichnung von Kanten $(e_1, ..., e_m)$ wird hier $(z_1, ..., z_m)$ benutzt, um eine eindeutige Trennung zur Messabweichung $(e_r; e_s)$ zu ermöglichen.

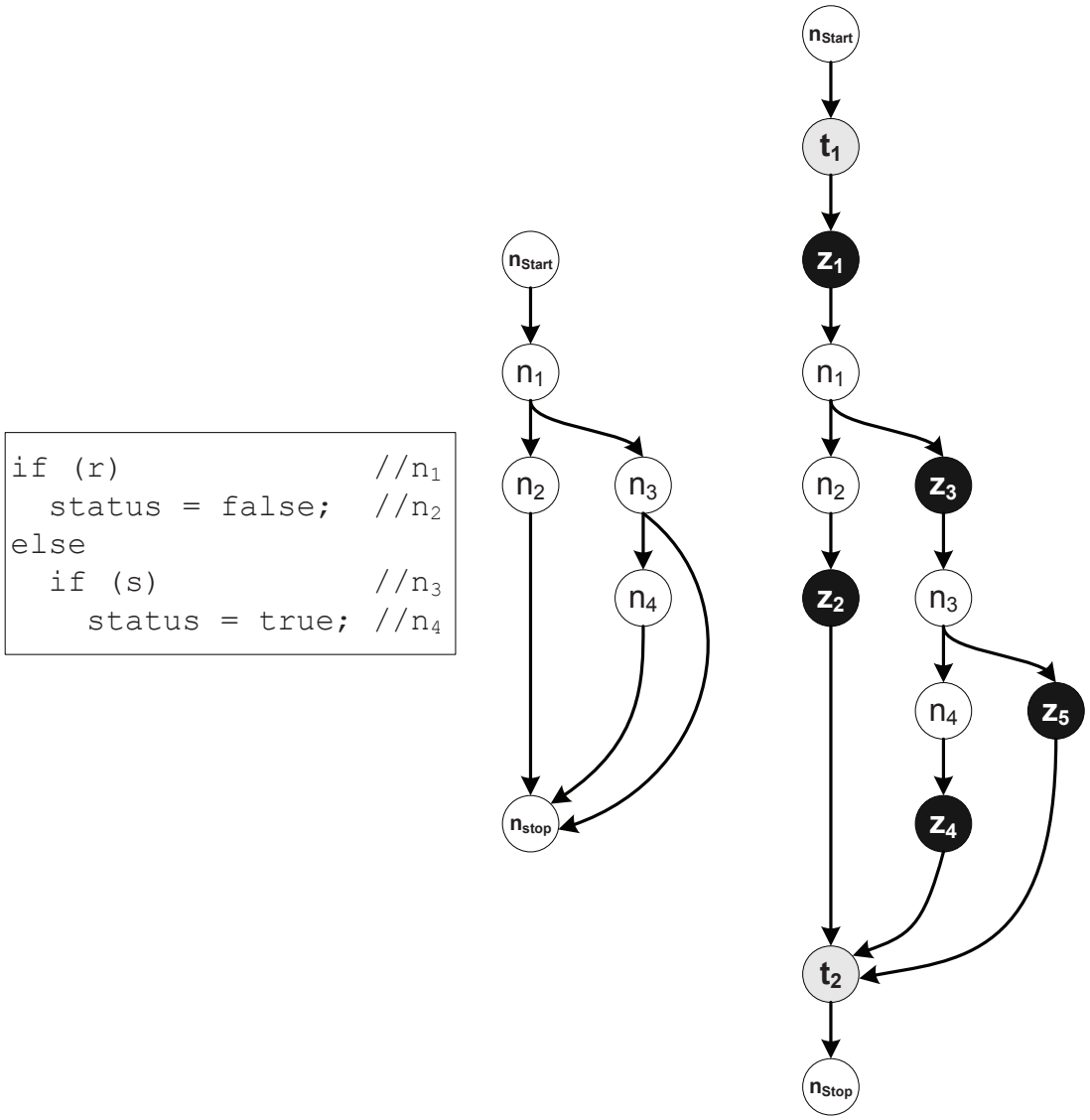


Abb. 4.4: Beispielprogramm mit Kontrollflussgraph und Instrumentierung

Als Ausgangsbeispiel soll das in Abb. 4.4 dargestellte Programm dienen. Der linke Graph zeigt den Kontrollfluss des Programms mit den Knoten $(n_{start}, n_1, n_2, n_3, n_4, n_{stop})$. Der rechte Kontrollflussgraph zeigt das Programm mit Zweiginstrumentierung $(z_1, z_2, z_3, z_4, z_5)$ und Messpunktinstrumentierung (t_1, t_2).

Bei der Ausführung des Programms mit Instrumentierung wird eine Sequenz von Instrumentierungsinformationen ausgegeben. Die Ausgabe der Instrumentierungsinformationen ist in Tabelle 4.3 als Pfadmessung beschrieben. Hieraus lassen sich der durchlaufene Programmpfad und die enthaltenen Messpunkte rekonstruieren. Bei der Pfadmessung werden zur Vereinfachung der Abbildung von Messpunkten auf die Programmpfade die Messpunktnummern mit gemessen.

Tabelle 4.3: Programmpfade für Beispielprogramm

Vorgabe			Ausgabe	Rekonstruktion	
Testfall	Eingabe	Ergebnis	Pfadmessung	Pfad	Messpunkte
n	$d_{n,i}$ {r,s}	$a_{n,i}$ {status}	$pm_{n,i}$	$p_{n,i}$	$m_{n,i}$
1	1,0	false	t_1, z_1, z_2, t_2	z_1, z_2	t_1, t_2
2	0,1	true	t_1, z_1, z_3, z_4, t_2	z_1, z_3, z_4	t_1, t_2
3	0,0	unverändert	t_1, z_1, z_3, z_5, t_2	z_1, z_3, z_5	t_1, t_2
4	1,1	false	t_1, z_1, z_2, t_2	z_1, z_2	t_1, t_2

Das Beispiel zur Pfadmessung erhebt keinen Anspruch darauf, dass diese vier Testfälle das Beispielprogramm ausreichend genug prüfen. Für die Aufstellung der Testfälle wurde keine explizite Testmethode gewählt, so sind z. B. keine falschen Eingaben berücksichtigt und keine Vorbedingungen (Ausgangssituation) vorgegeben worden. Im Testfall 3 kann daher das Ergebnis nur als unverändert gegenüber dem vorherigen Zustand beschrieben werden. Die genaue systematische Testfallerstellung mit entsprechenden Testmethoden ist bereits Bestandteil des Software-Modultests.

Fehlerbetrachtung Pfadmessung: Unter der Annahme, dass die Messabweichung nicht vollständig beschrieben werden kann, werden die Pfadverfolgung und die Laufzeitmessung in getrennten Messungen durchgeführt. Da durch die Instrumentierung zusätzliche Anweisungen in den Programmcode gelangen, ist der unbekannt bleibende Anteil der systematischen Messabweichung $e_{s,u}$ und die nicht genau feststellbare zufällige Messabweichung e_r der Laufzeitmessung ohne die zusätzlichen Instrumentierungen zur Pfadmessung kleiner. Es werden also zu jedem Messgegenstand eine Pfadmessung und eine Laufzeitmessung durchgeführt. Die Synchronisation zwischen Laufzeitmessung und dem dabei durchlaufenen Pfad erfolgt über die Belegung der Eingangs-, Zustands- und Ausgangssignale.

Messen von Einflussgrößen

Wie am Anfang dieses Abschnittes eingeführt, soll der Bezug zwischen der Prozesslaufzeit (mit dem Merkmalspaar $[d_{n,i}; \Delta t^j_{\mathrm{Pr}E,i}]$) und dem Programmpfad (mit dem Merkmalspaar $[d_{n,i}; p_{n,i}]$) über die Einflussgrößen (Eingabe $d_{n,i}$) hergestellt werden. Mit dem Software-Modultest und dem temporalen Test sind daher alle Eingangs- und Ausgangssignale sowie Zustandsgrößen des Moduls aufzuzeichnen. In Abb. 4.5 wird das Modul B einer Software-Funktion getestet, hierbei werden alle als Messsignal gekennzeichneten Größen aufgezeichnet. Das Aufzeichnen von Eingaben und Ausgaben ist Teil des Software-Modultest, dieser kann auf einem leichtzugänglichen Entwicklungssystem, wie einer virtuellen Testumgebung, oder auf einem Entwicklungsboard erfolgen (s. [13]). Daher kann dieser Teil des temporalen Prüfkonzeptes direkt aus der bisherigen Prüfstrategie (s. Abschn.2.5) verwendet werden.

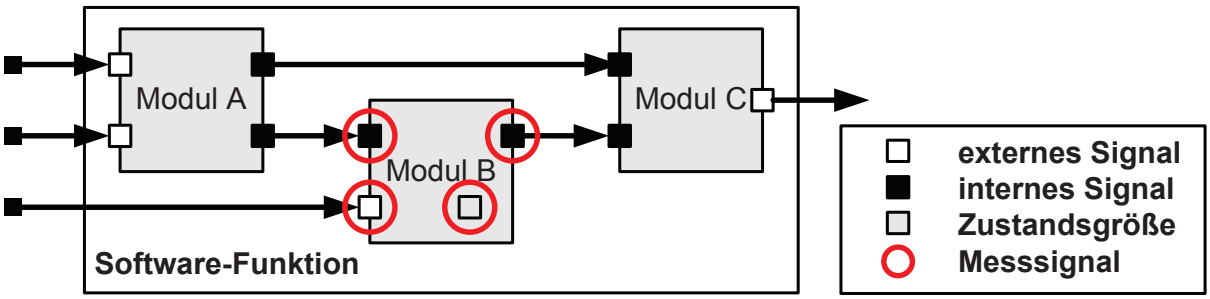


Abb. 4.5: Messsignale im Modultest

Ein weit größeres Problem stellt das Monitoring bzw. die Zugänglichkeit des Motorsteuergerätes (eingebettetes Echtzeitsystem) bei der Messung der Eingaben und der Prozesslaufzeit dar. Daher sollen nachfolgend die bestehenden Testverfahren berücksichtigt und deren Schnittstellen zum Steuergerät analysiert werden.

Das Messen von Signalen des Steuergerätes ist ein wesentlicher Bestandteil des Funktionstests und der Applikation von programmierbaren elektronischen Steuerungen. Durch Beobachtung von Messsignalen können die Auswirkungen von Parameteränderungen beurteilt und das Zusammenspiel der Komponenten eines Fahrzeugsystems bei der Ausführung einer bestimmten Fahrzeugfunktion optimiert werden.

Das Messen wird über Mess- und Kalibriersysteme vorgenommen, diese Systeme setzten sich aus einem Mess- und Kalibrierwerkzeug sowie einem Steuergerät mit Zusatzmesstechnik zusammen. Diese Steuergeräte werden allgemein als Applikations- oder Entwicklungssteuergeräte bezeichnet, die im Labor, am Prüfstand sowie im Fahrzeug eingesetzt werden können. Während der Ausführung des Programms muss das Mess- und Kalibriersystem die Erfassung von sich ändernden Signalen im Mikrocontroller - also die Messung von Größen, die im RAM des Mikrocontrollers abgelegt sind - synchron mit der Erfassung von zusätzlichen Signalen der Steuergeräteumgebung ermöglichen, hierzu zählt auch die Erfassung von Signalen des Fahrers oder der Umwelt. Die Messtechnik muss sicherstellen, dass die zeitliche Synchronisation von Signalen des Mikrocontrollers erhalten bleibt. Die Messtechnik, die dies leisten kann, wird im folgenden Abschnitt beschrieben.

4.4.2 Monitorsystem zur Erfassung von Ziel- und Einflussgrößen

Das hier beschriebene Monitorsystem ist im Wesentlichen für das Verstellen von Größen (Parametern) zur sogenannten Online-Kalibrierung konzipiert. Im Seriensteuergerät sind diese Parameter im nichtflüchtigen Speicher abgelegt und werden mit Ausnahme von adaptiven Parametern nicht verändert. Entwicklungssteuergeräte müssen mit zusätzlichem Speicher (Calibration RAM, CAL-RAM) ausgerüstet werden, die zu verstellenden Daten werden vom ROM- oder Flash-Bereich in einen freien RAM-Bereich kopiert. Parallel zum Steuergeräteprogramm kann ein externes Kalibrierwerkzeug auf den CAL-RAM zugreifen und die dort gespeicherten Parameter zur Laufzeit verändern. Der zusätzliche CAL-RAM wird über den Daten- und Adressbus des Mikrocontrollers angeschlossen. Bei einem parallelen Emulatortastkopf (ETK) [96] ist der CAL-RAM bereits integriert, muss also nicht wie bei alternativen Applikationsverfahren auf der Steuergeräteplatine untergebracht werden. Die Steuergerätedaten werden in zwei umschaltbaren Datenbereichen (Arbeits- und Referenzdatensatz) gehalten. Auf die emulierten Daten im CAL-RAM (Arbeitsseite) kann, im Unterschied zur Referenzseite (ROM- oder Flash-Bereich), während der Ausführung des Fahrprogramms durch das Kalibrierwerkzeug gearbeitet werden. Eine Unterbrechung der Ausführung des Programms, wie beim Arbeiten auf dem ROM- oder Flash-Bereich, ist nicht notwendig.

Bei Mikrocontrollern ohne externen Bus wird der CAL-RAM vom Chiphersteller in einer Entwicklungsvariante des Mikrocontrollers integriert. In diesem Fall kann mit einem seriellen ETK über eine Test- oder Debug-Schnittstelle des Mikrocontrollers auf den internen CAL-RAM zugegriffen werden. Die Erfassung von Messgrößen aus dem Steuergerät wird durch die ETK-Schnittstelle mit bis zu 100 MBit/s unterstützt. Hunderte von Steuergerätemessgrößen können zeit- und winkelsynchron in bis zu 32 Rechenrastern zeitgleich erfasst werden.

Im Wesentlichen sind damit drei Schnittstellen zu identifizieren, die für die Ausgabe der Laufzeitmessdaten infrage kommen. In Abb. 4.6 sind die verschiedenen Schnittstellen zwischen Kalibrierwerkzeug und Mikrocontroller zum Zugriff auf den RAM-/bzw. CAL-RAM-Bereich dargestellt. Das Kalibrierwerkzeug kann entweder über serielle Schnittstellen oder den parallelem Bus des Mikrocontrollers auf den RAM-Bereich zugreifen.

- Die seriellen Schnittstellen über K-Leitung [92] oder CAN [91] werden auch im Serien-steuergerät eingesetzt und verlangen keine Hardwareanpassung. Diese Schnittstellen sind jedoch in ihrer Übertragungsleistung begrenzt, zudem werden sie auch für die Offboard-Diagnosekommunikation oder die Onboard-Kommunikation eingesetzt, was die verfügbare Übertragungsleistung weiter einschränkt. Der Mikrocontroller wird bei der Übertragung von Daten belastet, da die Kommunikation zwischen Mikrocontroller und Werkzeug durch Software-Komponenten realisiert wird, die zusätzliche Ressourcen, also Laufzeit und Speicher, belegen.
- Eine weitere serielle Schnittstelle besteht in der Entwicklungsphase über die NEXUS- [93] oder JTAG- [94] Schnittstelle, die eigentlich für Download und Debugging der Software zur Verfügung steht. Die Hardware des Steuergerätes muss nicht angepasst werden, lediglich die Hardware des seriellen Emulatortastkopf (ETK) [95] muss an diese Schnittstellen angebunden werden. Die Übertragungsleistung ist recht hoch und hängt von der Leistungsfähigkeit der Debugging-Schnittstelle ab. Die Kommunikation zwischen Mikrocontroller und Werkzeug wird durch Hardware realisiert, der Einfluss dieses Verfahrens auf die Laufzeit der Fahrzeugfunktionen ist gering. Diese Schnittstelle wird meist dann für Applikationssteuergeräte verwendet, wenn der Mikrocontroller des Steuergeräts keinen externen Daten- und Adressbus zur Verfügung stellt.

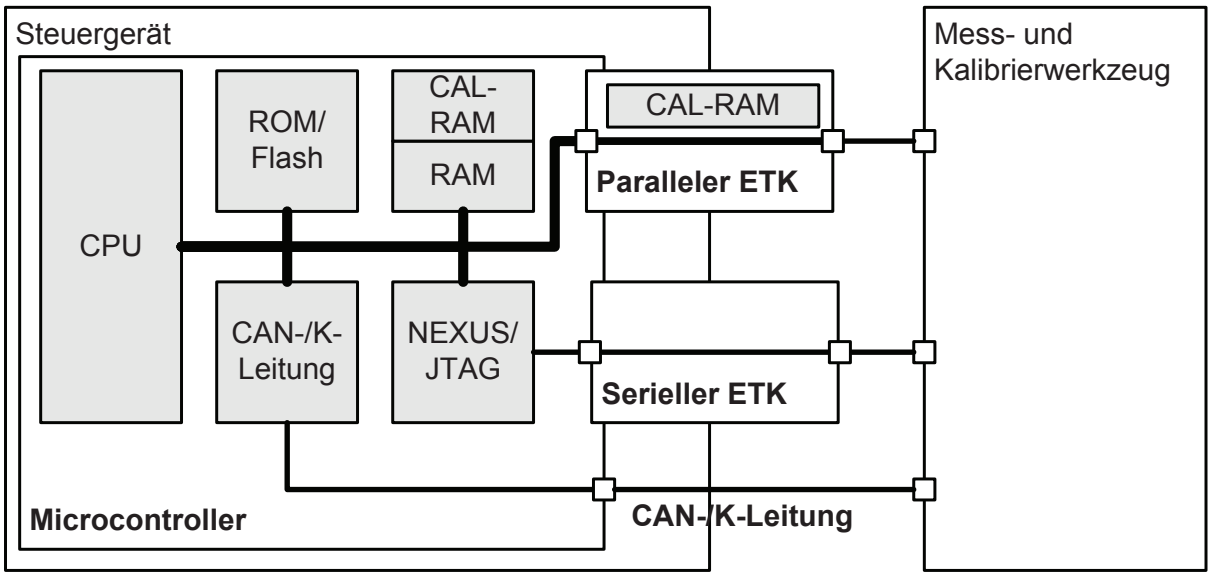


Abb. 4.6: Monitorsystem

- Der Zugriff über den parallelen Bus des Mikrocontrollers ist nur in der Entwicklungsphase möglich. Hierfür muss der Mikrocontroller über einen externen Daten- und Adressbus verfügen, an den der parallele ETK [96] angeschlossen wird. Die Übertragungsleistung übertrifft die der beiden anderen Schnittstellen. Es ist ein unabhängiger CAL-RAM-Zugriff von Mikrocontroller und Werkzeug möglich. Der Einfluss dieses Verfahrens auf die Laufzeit der Fahrzeugfunktionen ist aufgrund des unabhängigen RAM-Zugriffs sehr gering.

Fehlerbetrachtung Monitorsystem: Bei allen drei Verfahren hängt die Belastung des Mikrocontrollers von der Arbeit des CAL-RAM-Managements ab. Hierzu gehört die Umschaltung zwischen Arbeits- und Referenzseite, die vom Kalibrierwerkzeug gesteuert wird, das Speichern und Kopieren von Daten zwischen den beiden Seiten sowie ein Überwachungskonzept bei unplausiblem Verhalten.

Für den Datenzugriff muss dem Mess- und Kalibriersystem eine Adressinformation übergeben werden, an welcher Stelle im Speicher die Mess- und Kalibrierdaten liegen, dies erfolgt über die Beschreibungsdatei im Format ASAM-MCD 2MC (a2l) (s. Abschn. 2.1.2). Nach Auswahl einer Größe aus der a2l-Datei im Mess- und Kalibrierwerkzeug kann diese in einem definierten Rechenraster (Task τ_i) des Steuergerätes aufgenommen und an das Werkzeug ausgegeben werden. In Abb. 4.7 wird dies als „Übertragungsroutine für Messdaten“ bezeichnet, diese Routine läuft unabhängig vom Messprozess ab. Die Übertragungsroutine stört damit die Laufzeit des Messprozesses nicht.

Während der Laufzeitmessung darf keine Applikation (Kalibrierung der Software), d. h. eine Veränderung des Funktionsverhaltens über Applikationsparameter, erfolgen. Der Zugriff des Programms auf Parameter, das sind Kennwerte, Kennlinien und Kennfelder der Software-Funktion, darf nur auf die Referenzseite erfolgen. Dieses Vorgehen verhindert, dass die Applikationsparameter aus dem CAL-RAM-Bereich geladen werden.

Die gemessenen Prozesslaufzeiten werden damit nicht durch höhere Zugriffs- und Transferzeit sowie Eingriffe des CAL-RAM-Managements beeinflusst. Als Messabweichung, hervorgerufen durch die Ausgabe der Laufzeitmessdaten an das Monitorsystem, muss das Zwischenspeichern der Messdaten im RAM (vor der Übertragung) betrachtet werden. Die hierfür benötigte Laufzeit für das Speichern der Daten kann der Messpunktlaufzeit zugerechnet werden und wird in Abb. 4.7 als Messabweichung je Messpunkt (MP) bezeichnet.

4.4.3 Testumgebung

Der grundlegende Messaufbau (s. Abb. 4.8) setzt sich aus drei Komponenten zusammen. Die erste Komponente ist das Steuergerät, für die die zu testende Software-Funktion erstellt wurde. Dort wird das zu untersuchende Programm ausgeführt und gemessen. Das Programm wird unter realen Bedingungen entweder im Labor (z. B. am HiL-Simulator), am Prüfstand oder im Fahrzeug zur Ausführung gebracht.

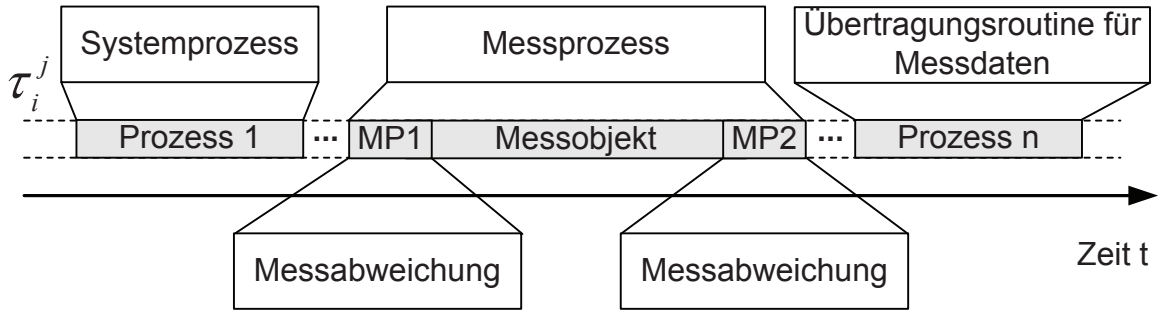


Abb. 4.7: Messprozess im Task *i* bei der *j*-ten Ausführung (τ_i^j)

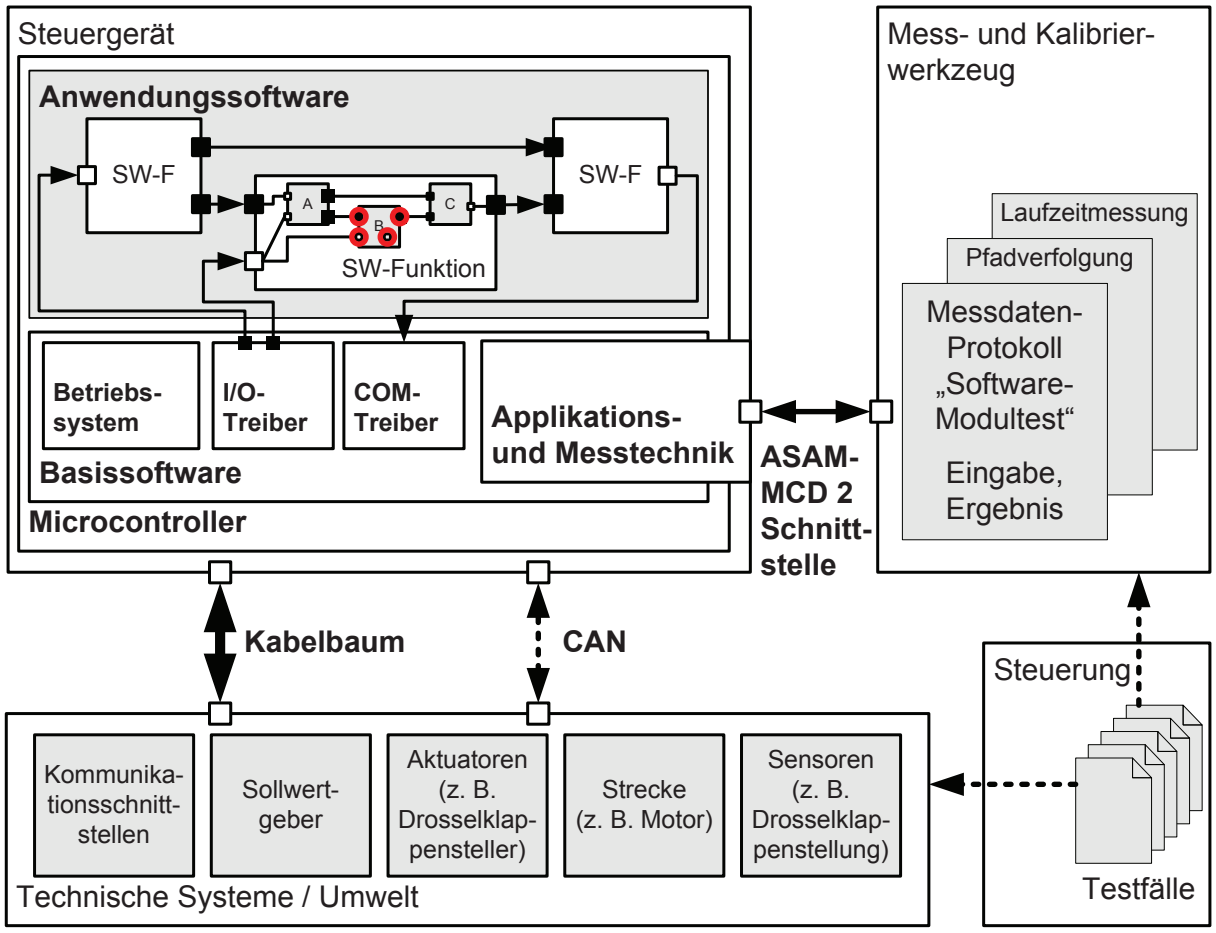


Abb. 4.8: Testumgebung

Die zweite Komponente ist die Steuerung, dies kann ein Tester sein, der die vorgegebenen Eingaben (Testfälle) vornimmt. Die Verbindung zum Steuergerät wird z. B. über die Sollwertgeber ermöglicht. Das Steuersystem kann auch als Bestandteil der Testautomatisierung eines HiL-Systems durch einen Steuerrechner abgebildet werden. Die dritte Komponente ist die Messeinrichtung, bestehend aus dem Mess- und Kalibrierwerkzeug sowie der Zusatzmesstechnik auf dem Steuergerät. Die Messeinrichtung protokolliert die gemessenen Signale.

4.5 Testprozess zur Pfadverfolgung und Laufzeitmessung

Der in diesem Kapitel vorgestellte Teil des temporalen Prüfkonzeptes liefert als neuen Ansatz einen in den bestehenden Testprozess integrierten temporalen Test, der als „Strategie zum temporalen Test“ formuliert wurde. Als Zwischenergebnis werden hier, als Folge von Tätigkeiten zur Datengewinnung, Programmpfadinformationen und Prozesslaufzeitinformationen zu jedem Testfall ausgegeben, ohne den bestehenden Testaufwand zu erhöhen. Dies gelingt, in dem Synergien zum bestehenden funktionalen Test genutzt werden und eine Wiederverwendung von bereits erstellten Testfällen aus dem Software-Modultest als temporaler Regressionstest vorgeschlagen wird.

Bevor die temporale Prüfung auf Basis der ausgegebenen Messdaten im Kapitel 5 vorgestellt wird, soll anhand des Bespielprogramms (s. Abb. 4.4) die Integration des temporalen Tests in den Testprozess bis zur Erzeugung der Testprotokolle zusammengefasst dargestellt werden.

Pfadverfolgung

Im ersten Schritt des Testprozesses (s. Abb. 4.9) wird der Programmcode mit den Instrumentierungen zur Pfadmessung versehen. Der Compiler erzeugt daraus im nächsten Schritt den Objektcode. Der Programmcode wird vom Benutzer im Software-Codetest (s. Abschn. 2.6.2) einer statischen Codeanalyse unterzogen, um Strukturinformationen über die Zweige des Programms zu erhalten. Diese Informationen dienen dazu, im Rahmen der Auswertung der Messergebnisse, die Kriterien von Anweisungs-, Zweig- und Pfadabdeckung zu prüfen. Weiterhin werden vom Benutzer mithilfe der Schnittstellenspezifikation die zu erfassenden Signale ausgewählt, um die Eingabedaten und Ergebnisse zu erfassen. Alle externen Signale und Zustandsgrößen müssen hier aufgezeichnet werden. Die Eingaben zum Stimulieren des Testobjektes werden entsprechend den funktions- und strukturorientierten Testverfahren als Testspezifikation erstellt (s. Tabelle 4.4). Bei der Testdurchführung erfolgt die Stimulation des Testobjektes in Abhängigkeit von der Stimulierung der Eingangssignale. In diesem Schritt wird auch die Pfadmessung durchgeführt.

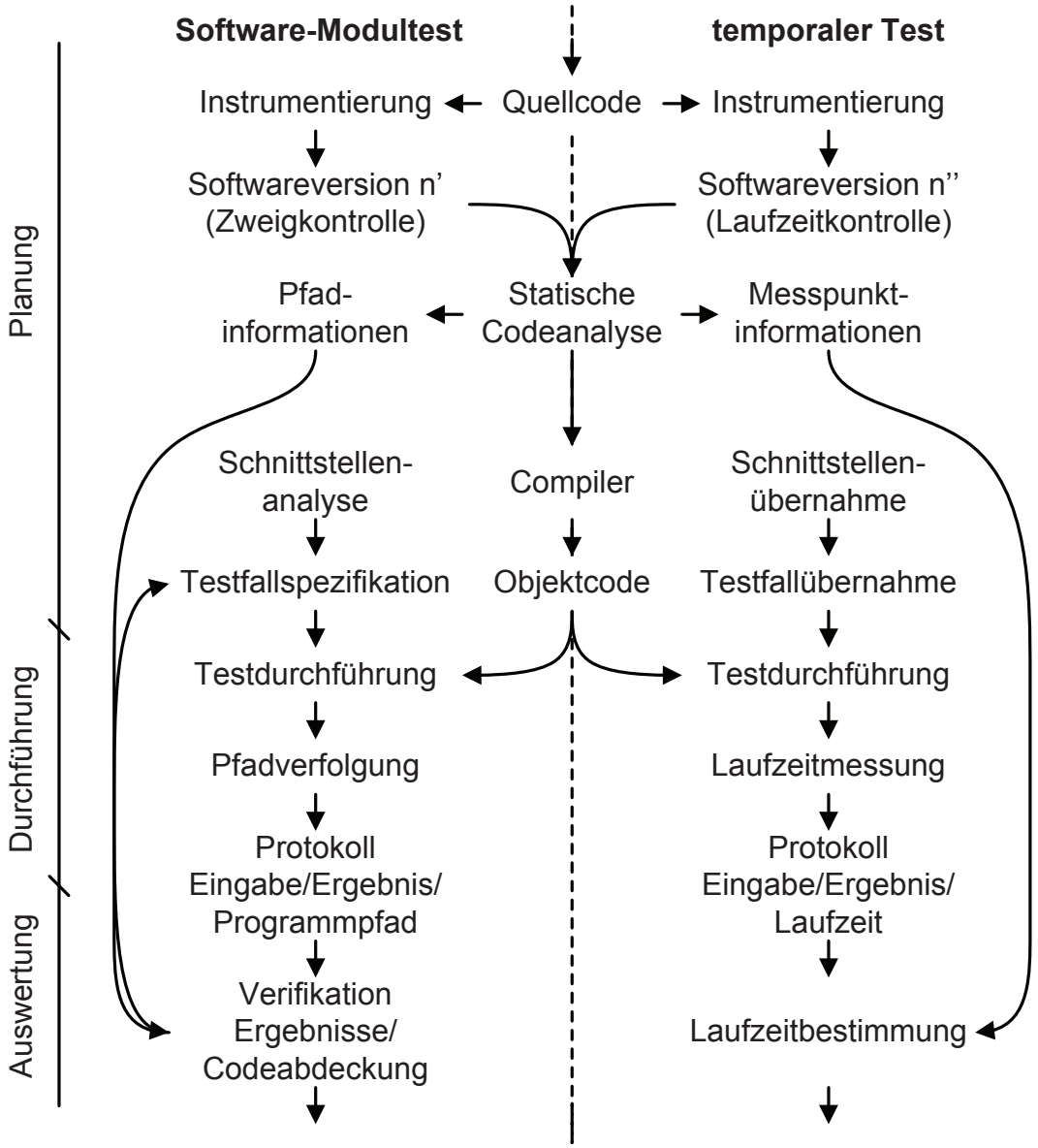


Abb. 4.9: Testprozess zur Pfadverfolgung und Laufzeitmessung

Die als Testprotokoll aufgezeichnete Sequenz aus Messpunkten und Zweigen wird als Signatur bezeichnet. Im Rahmen der Auswertung werden die Ergebnisse des Software-Modultests verifiziert und die Anweisungs-, Zweig- und Pfadabdeckung entsprechend den Testzielen bewertet. Die Tabelle 4.4 zeigt das Ergebnis der Pfadmessung als Testprotokoll zur Testspezifikation für das Beispielprogramm.

Tabelle 4.4: Testspezifikation und Testprotokoll Pfadverfolgung

Testspezifikation			Testprotokoll Pfadverfolgung			
Testfall	Eingabe[a]	Ergebnis[b]	Ausführung	Eingabe[a]	Ergebnis[b]	Signatur
1[c]	1,0	false	1	1,0	0	t_1, z_1, z_2, t_2
2	0,1	true	2	0,1	1	t_1, z_1, z_3, z_4, t_2
3	0,0	true	3	0,0	1	t_1, z_1, z_3, z_5, t_2
4	0,1	true	4	0,1	1	t_1, z_1, z_3, z_4, t_2
5	1,1	false	5	1,1	0	t_1, z_1, z_2, t_2
6	0,0	false	6	0,0	0	t_1, z_1, z_3, z_5, t_2
7	0,1	true	7	0,1	1	t_1, z_1, z_3, z_4, t_2
*	*	*	*	*	*	*
n	$d_{n,i}$	$a_{n,i}$	j	$d_{n,i}$	$a_{n,i}$	$pm_{n,i}$

a - Eingabe {r,s}, b - Ergebnis {status}, c - Grundzustand bzw. Ausgangssituation des Tests

Laufzeitmessung

Vor der Messung wird das Programm mit den Instrumentierungen zur Laufzeitmessung versehen (s. Abb. 4.9). Die statische Codeanalyse liefert hier Strukturinformationen über die enthaltenen Messpunkte. Mit diesen Informationen werden im Rahmen der Datenanalyse die Laufzeiten zwischen zwei Messpunkten dem jeweiligen Prozess zugeordnet. Es werden bei der Testdurchführung die gleichen Signale des Moduls erfasst wie beim vorhergehenden Software-Modultest. Die Eingaben zum Stimulieren des Messobjektes werden aus der Testspezifikation zum Software-Modultest (Pfadverfolgung) übernommen. Mit der Testdurchführung erfolgt die Laufzeitmessung. Das Ergebnis des temporalen Tests ist das Testprotokoll mit den Werten der einzelnen Analysemarken. Die im Testprotokoll aufgezeichnete Sequenz aus Messpunktnummer und Messpunktwert wird als Messung bezeichnet. Der Vergleich aus Ist-Laufzeit mit der maximal zulässigen Laufzeit wird als Messergebnisprotokoll bezeichnet. Die Ist-Laufzeit wird aus der Differenz der Messwerte berechnet (s. Gl. 4.1) und der Umrechnung in eine Zeit (s. Gl. 4.2).

$$c^j = t^j_{n+1} - t^j_n \tag{4.1}$$

Die Gl. 4.2 folgt dem in Gl. 2.11 beschriebenen allgemeinen Fall der Zeitberechnung aus einem Zählerstand.

$$\Delta t^j_E = c^j * t_{Timerzyklen} \tag{4.2}$$

In Tabelle 4.5 sind für das Beispielprogramm das Testprotokoll der Laufzeitmessung und das Messergebnisprotokoll dargestellt. Hier wird noch keine Betrachtung der systematischen Messabweichung e_s und der zufälligen Messabweichung e_r vorgenommen.
Die abschließende Bewertung, d. h. die Prüfung, ob das Messobjekt die geforderte Laufzeitgrenze einhält, erfolgt im Rahmen der Datenanalyse. Das gezeigte Messergebnisprotokoll enthält bereits einen Teil der Datenanalyse. Bei der Datenanalyse werden die aus den Messungen bestimmten Laufzeiten nach gemessenen pfadabhängigen WCET durchsucht und bewertet. Dafür müssen bei der Auswertung die Pfade mit den Laufzeiten kombiniert werden. Die Datenanalyse wird im folgenden Kapitel beschrieben.

Tabelle 4.5: Testprotokoll Laufzeitmessung und Messergebnisprotokoll

Testprotokoll Laufzeitmessung				Messergebnisprotokoll		
Ausführung	Eingabe[a]	Ergebnis[b]	Messung	Mess-ergebnis[d]	max. zul. Laufzeit[e]	Vergleichs-ergebnis
1[c]	1,0	0	t_1 4072634 t_2 4072689	0,733	10	kleiner
2	0,1	1	t_1 4811128 t_2 4811186	0,773	10	kleiner
3	0,0	1	t_1 5568742 t_2 5568796	0,719	10	kleiner
4	0,1	1	t_1 6323640 t_2 6323698	0,773	10	kleiner
5	1,1	0	t_1 7066252 t_2 7066307	0,733	10	kleiner
6	0,0	0	t_1 7814594 t_2 7814648	0,719	10	kleiner
7	0,1	1	t_1 8575822 t_2 8575880	0,773	10	kleiner
*	*	*	*	*	*	*
j	$d_{n,i}$	$a_{n,i}$	$t_{n,i}$	$\Delta t^{j}_{\mathrm{Pr}E,i}$	$\Delta t_{A\max}$	

a - Eingabe {r,s}, b - Ergebnis {status}, c - Grundzustand bzw. Ausgangssituation der Messung, d - in µs bei $t_{\mathrm{Timerzyklen}}$ = 13,33 ns, e - in µs

5 Datenanalyse

Der Messprozess, wie er in dieser Arbeit vorgeschlagen wird, soll um eine statische Auswertung der Messdaten ergänzt werden. Dazu ist es notwendig, die bei den Messungen anfallenden Daten in einer effizienten Form für die Analyse bereitzustellen. Die Reproduzierbarkeit der Testergebnisse muss ebenso möglich sein. Um diese Anforderungen zu erfüllen, wurde im Rahmen dieser Arbeit ein Datenanalyse- und Dokumentationskonzept entworfen und in ein Analysesystem implementiert, welches die Aufgabe der Messdatenorganisation (Datenbank), der Messdatenauswertung (Datenanalyse) sowie die Testdokumentation übernimmt (s. Abb. 5.1).

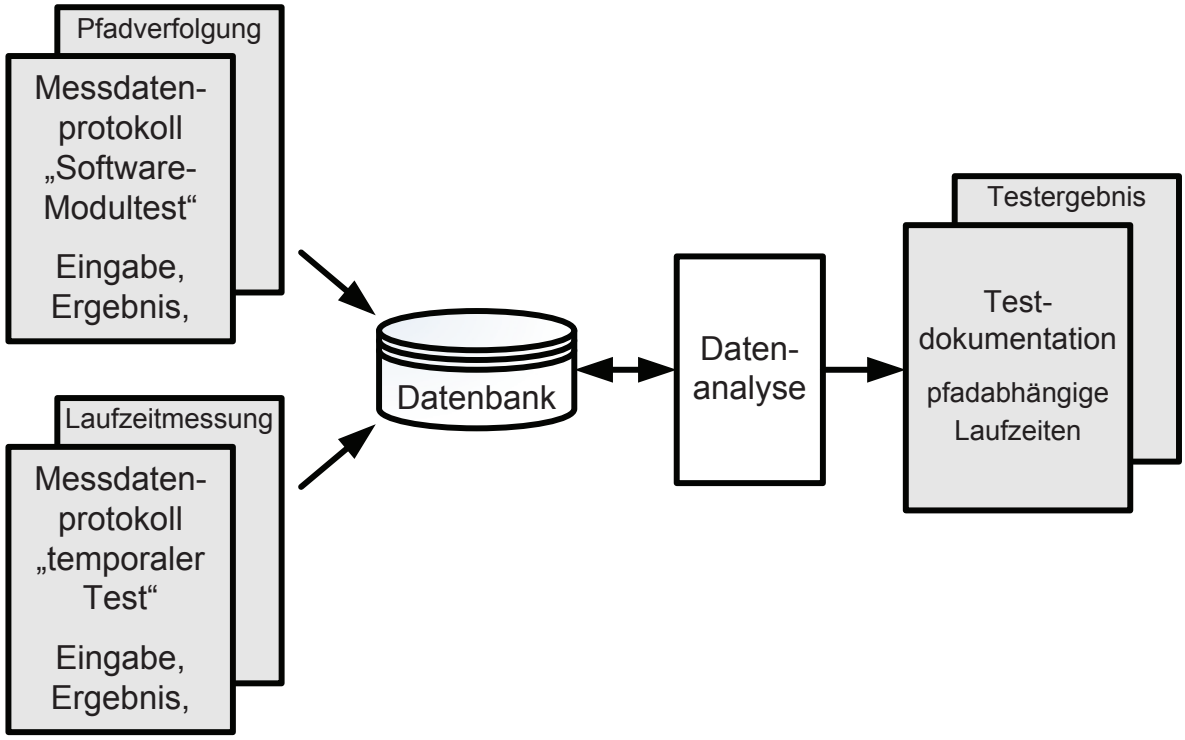


Abb. 5.1: Aufbau Analysekonzept

Die beim funktions- und strukturorientierten Software-Modultest anfallenden Messdaten (Eingabe, Ergebnis und Pfadverfolgung) werden als Referenzdaten aufgezeichnet. Diese Messdaten werden für eine Datenanalyse zusammen mit den Messdaten aus dem temporalen Test (Eingabe, Ergebnis und Laufzeitmessung) organisiert. Anschließend wird auf Basis dieser Daten die pfadabhängige Laufzeitbestimmung vorgenommen.

5.1 Programmpfadorientierte Datenanalyse

Bei Redundanzen zwischen den Daten der Pfadverfolgung und der Laufzeitmessung lassen sich Beziehungen automatisch bestimmen und für die Komplettierung der Datensätze (Datenfusion) nutzen. Bei Übereinstimmung der Einträge von Eingabe $d_{n,i}$ und Ergebnis $a_{n,i}$ in den

beiden Messdatenprotokollen können Pfad und Laufzeit kombiniert werden. Ausgangspunkt der Datenanalyse bilden die Datenquellen der Pfadverfolgung (Kapitel 4 - Tabelle 4.4) und der Laufzeitmessung (Kapitel 4 - Tabelle 4.5).

5.1.1 Messdatenorganisation und Datenfusion

Die Grundzüge der Messdatenorganisation und der Datenfusion werden im Folgenden dargestellt. Aufgrund der Zweckmäßigkeit des Entity-Relationship-Modells (ER-Modell bzw. ERM), das die grundlegende Tabellen- und Beziehungsstruktur einer Datenbank visualisiert, wurde hier die Chen-Notation [97] zur Beschreibung der Zusammenhänge gewählt.

Die Organisation der Daten erfolgt innerhalb des Testprotokolls, das alle relevanten Informationen zum Test verwaltet, wie Name der Fahrzeugfunktion, Pfad zu den Quelldaten (Funktionsmodell) und eine Versionsinformation. Zu jedem Testprotokoll können mehrere Testfälle abgelegt werden, diese umfassen Informationen wie Funktionsname, Datum, Tester, Steuergerät, CPU, Speicher, Cache, Pfad als Referenz zu den Quelldaten (Quellcode, Objektcode, Messdaten), Testmethode und Leistungsmerkmale. Die Testfälle eines Protokolls beinhalten die gemessenen Pfade, diese setzen sich aus den Eingangsbelegungen und den Pfaden zusammen. Bei den Eingangsbelegungen wird nicht differenziert zwischen Eingabe und Ergebnis, dies geschieht ausschließlich über die Signalnamen. Die Messung enthält die Informationen von Eingangsbelegung, Messpunktnummer und Messwert. Im Mittelpunkt der weiteren Auswertung steht die Ermittlung der programmpfadabhängigen Prozesslaufzeit. Die Objekte der Datenbank (Entitäten) sind Testprotokoll, Testfall, Pfad, Messung und Eingangsbelegung. Die Abb. 5.2 zeigt das ER-Modell für die Pfad- und Messdatenorganisation. Die Attribute (Eigenschaften) wurden hier weggelassen, um eine übersichtlichere Darstellung zu ermöglichen.

Aus dem ER-Modell ergeben sich die folgenden Beziehungen der Entitäten:

- 1 bis m Testprotokolle beschreiben 0 bis n Testfälle.
- 1 bis n Testfälle enthalten 1 bis m Pfade.
- 0 bis n Messungen gehören zu einem Testfall.
- 0 bis m Messungen gehören zu einem Pfad.
- Eine Messung wird stimuliert durch 1 bis n Eingangsbelegungen.
- Ein Pfad wird stimuliert durch 1 bis m Eingangsbelegungen.

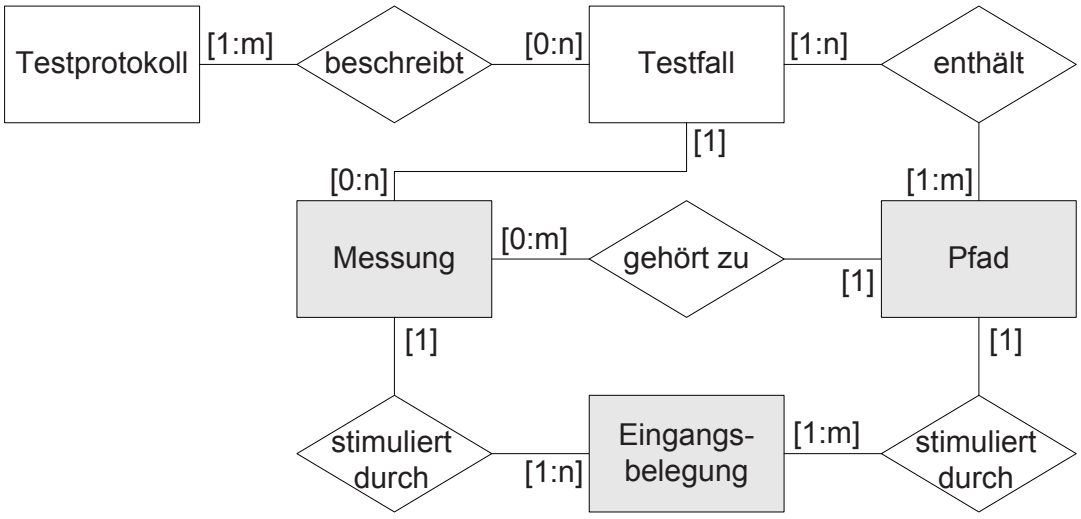


Abb. 5.2: ER-Modell Pfad- und Messdatenorganisation

5.1.2 Programmpfadanalyse

Zusätzlich zu den Referenzdaten (Belegung von Eingabe $d_{n,i}$ und Ergebnis $a_{n,i}$) werden die gemessenen Pfade $p_{n,i}$ (repräsentiert durch die Signatur $pm_{n,i}$) aufgezeichnet, die bei der Testdurchführung durchlaufen wurden. Zu jedem gemessenen Programmpfad $p_{n,i}$ des Prozesses *i* für die Ausführung durch den Testfall *n* stehen folgende Informationen (s. Abb. 5.3) in den Entitäten Pfad und Eingangsbelegung zur Verfügung.

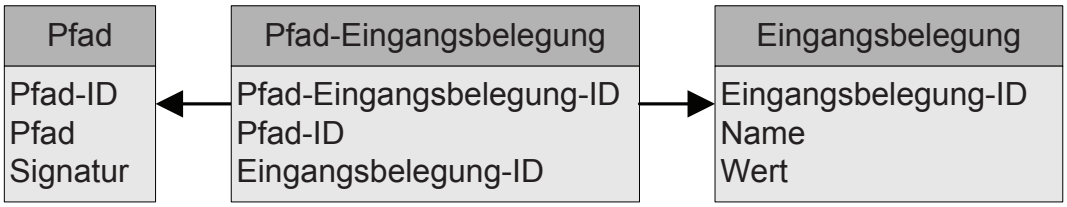


Abb. 5.3: Pfad zu Eingangsbeziehung

- Die Entität Pfad enthält:
  - die Pfadsignatur $pm_{n,i}$ in Form der Sequenz der Zweige $(z_1,...,z_m)$ und Messpunkte $(t_1,...,t_m)$,
  - den Pfad $p_{n,i}$ als Sequenz der Zweige $(z_1,...,z_m)$ (rekonstruiert aus $pm_{n,i}$) und
  - die Pfad-ID als eindeutige Nummerierung für jeden unterschiedlichen Programmpfad.
- Die Entität Eingangsbelegung enthält:
  - die Namen der Eingangssignale $d_{n,i}$ und Ausgangssignale $a_{n,i}$,
  - die Werte der Eingangssignale $d_{n,i}$ und Ausgangssignale $a_{n,i}$ und
  - die Eingangsbelegung-ID als eindeutige Nummerierung jeder unterschiedlichen Eingangs- und Ausgangsbelegung.

Die Verknüpfung erfolgt über die Beziehung Pfad-Eingangsbelegung.

- Die Beziehung Pfad-Eingangsbelegung beschreibt:
  - die Beziehung zwischen Pfad-ID und Eingangsbelegung-ID.

Zur Veranschaulichung sind für das Beispielprogramm aus Kapitel 4 (s. Tabelle 4.4) im Folgenden die beiden Entitäten und deren Beziehung in Tabelle 5.1 präsentiert.

Tabelle 5.1: Entitäten und Beziehung von Pfad-Eingangsbelegung

Pfad		
ID	Pfad	Signatur
1	z_1, z_2	t_1, z_1, z_2, t_2
2	z_1, z_3, z_4	t_1, z_1, z_3, z_4, t_2
3	z_1, z_3, z_5	t_1, z_1, z_3, z_5, t_2

Pfad-Eingangsbelegung		
ID	Pfad-ID	Eing.-ID
1	1	1
2	1	2
3	1	3
4	1	5
5	2	4
6	2	5
7	2	6
8	3	2
9	3	3
10	3	4
11	3	6

Eingangsbelegung		
ID	Name	Wert
1	r	1
2	s	0
3	status	0
4	r	0
5	s	1
6	status	1

Aus diesen Daten lassen sich die durchlaufenen Programmpfade mit den zugehörigen Eingaben und Ergebnissen rekonstruieren. Zusätzlich lässt sich die Strukturabdeckung anhand der durchlaufenen Zweige zu der aus der statischen Codeanalyse erstellten Gesamtmenge von Zweigen beurteilen. Abhängig vom Resultat der Verifikation der Eingabe- und Ergebnisdaten sowie der Testabdeckung müssen Maßnahmen entsprechend der dynamischen Prüfstrategie ergriffen werden. Diese sind wie bereits beschrieben nicht Gegenstand der Arbeit. Ergebnis der Programmpfadanalyse und damit des Software-Modultests ist eine Testabdeckung nach den Vorgaben der Prüfstrategie. Dies beinhaltet eine vollständige Zweigabdeckung.

5.1.3 Laufzeitanalyse

In einer zweiten Testdurchführung erfolgt eine Messung der Laufzeiten zu den Referenzdaten aus dem funktions- und strukturorientierten Test. Die Datenanalyse der Messdaten wird nachfolgend als pfadabhängige Laufzeitbestimmung beschrieben. Das Ziel der pfadabhängigen Laufzeitbestimmung ist, zu jedem gemessenen Pfad eines Programms eine Laufzeit ermitteln zu können. Die Datenanalyse besteht aus einer zweistufigen Aufbereitung der Laufzeitmessdaten. Der erste Verfahrensschritt widmet sich der Beschreibung und der Darstellung der Messdaten. Der anschließende Verfahrensschritt befasst sich mit dem Auffinden von Strukturen.

Messdatenaufbereitung: Die Analyse besteht zum einen aus der Bestimmung der gemessenen Prozesslaufzeit $\Delta t^{j}_{\mathrm{Pr}E,i}$ des Prozesses *i* für die *j*-te Ausführung. Die Prozesslaufzeit $\Delta t^{j}_{\mathrm{Pr}E,i}$ wird aus der Differenz zweier Messpunktwerte gebildet, dem Prozessstart $t^{j}_{\mathrm{Pr}S,i}$ und dem Prozessabschluss $t^{j}_{\mathrm{Pr}C,i}$. Von jedem gemessenen Zeitpunkt t^{j}_{i} des Prozesses *i* im Testfall *n* stehen die nachfolgenden Informationen zur Verfügung (s. Abb. 5.4).

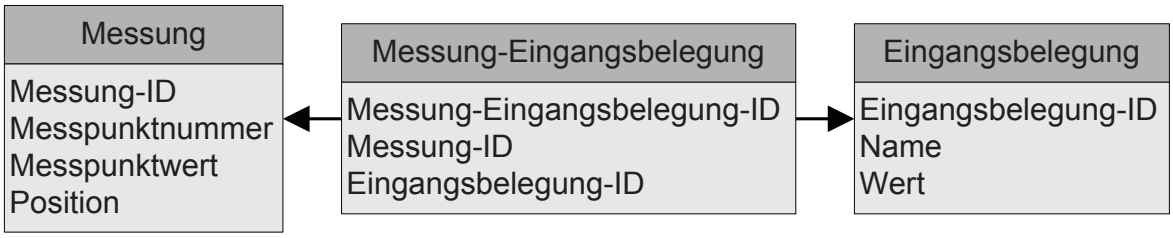


Abb. 5.4: Messung zu Eingangsbeziehung

- Die Entität Messung enthält:
  - die Laufzeitmessung t^{j}_{i} als Sequenz aus Messpunktnummer $(t_1,...,t_m)$ und Messpunktwert,
  - die Position als eine eindeutige Nummerierung unter der für jede Messung *j* (Ausführung) die enthaltenen Messpunktnummern und Messpunktwerte zusammengefasst werden und
  - die Messung-ID als eindeutige Nummerierung für jede Messpunktnummer mit Messpunktwert.
- Die Entität Eingangsbelegung enthält:
  - die Namen der Eingangssignale $d_{n,i}$ und Ausgangssignale $a_{n,i}$,
  - die Werte der Eingangssignale $d_{n,i}$ und Ausgangssignale $a_{n,i}$ und
  - die Eingangsbelegung-ID als eindeutige Nummerierung für jede unterschiedliche Eingangs- und Ausgangsbelegung.

Die Verknüpfung erfolgt über die Beziehung Messung-Eingangsbelegung.

- Die Beziehung Messung-Eingangsbelegung beschreibt:
  - die Beziehung zwischen Messung-ID und Eingangsbelegung-ID.

Zur Veranschaulichung der Zusammenhänge wird wieder auf das Beispielprogramm aus Kapitel 4 zurückgegriffen. In Tabelle 5.2 sind die zugehörigen Entitäten und deren Beziehung dargestellt.

Tabelle 5.2: Entitäten und Beziehung von Messung-Eingangsbelegung

Messung			
ID	MP-Nr.	MP-Wert	Position
1	t_1	4072634	0
2	t_2	4072689	0
3	t_1	4811128	1
4	t_2	4811186	1
5	t_1	5568742	2
6	t_2	5568796	2
7	t_1	6323640	3
8	t_2	6323698	3
9	t_1	7066252	4
10	t_2	7066307	4
11	t_1	7814594	5
12	t_2	7814648	5
13	t_1	8575822	6
14	t_2	8575880	6

Messung-Eingangsbelegung		
ID	Messung-ID	Eing.-ID
1	1	1
2	1	2
3	1	3
4	2	1
5	2	2
6	2	3
7	3	4
8	3	5
9	3	6
10	4	4
11	4	5
12	4	6
13	5	2
14	5	4
*	*	*

Eingangsbelegung		
ID	Name	Wert
1	r	1
2	s	0
3	status	0
4	r	0
5	s	1
6	status	1

Darstellung der Messdaten: Dieser Schritt enthält neben der reinen Visualisierung der Laufzeitmessdaten auch die Berechnung der Taktzyklendifferenz nach Gl. 4.1. Bis zu diesem Verfahrensschritt stehen die Laufzeitmessdaten in Form einer tabellarischen Aufzählung zur Verfügung, die für den Benutzer nur wenig Rückschlüsse und Interpretationsmöglichkeiten bieten. Hierfür besser geeignet sind graphische Darstellungen in Form von Verlaufskurven, Diagrammen und Häufigkeitsverteilungen, die zusätzliche Kenngrößen wie zum Beispiel den Mittelwert oder die gemessene BCET und WCET angeben können. Trägt man die Laufzeitmesswerte (Taktzyklendifferenz zweier Messpunkte) als Ordinaten und die Zeitpunkte der Messung t=0,1,2,..,j als Abszissen in ein kartesisches Koordinatensystem ein, ergibt sich Abb. 5.5. Die Abbildung stellt die Verlaufskurven für das Beispielprogramm dar, indem die Testfälle 1 bis 7 zyklisch wiederholt werden, insgesamt werden 497 Messungen aufgenommen. Bedingt durch kontrollierbare und unkontrollierbare Einflüsse auf dem Messvorgang lassen sich unterschiedliche Programmlaufzeiten feststellen. Soll die Laufzeit eines Programms möglichst exakt festgestellt werden, müssen diese Einflüsse in ihrer Wirkung bestimmbar bzw. ausschaltbar sein. Nachfolgend werden die Möglichkeiten diskutiert, um diese Einflüsse bei der Messung zu beherrschen.

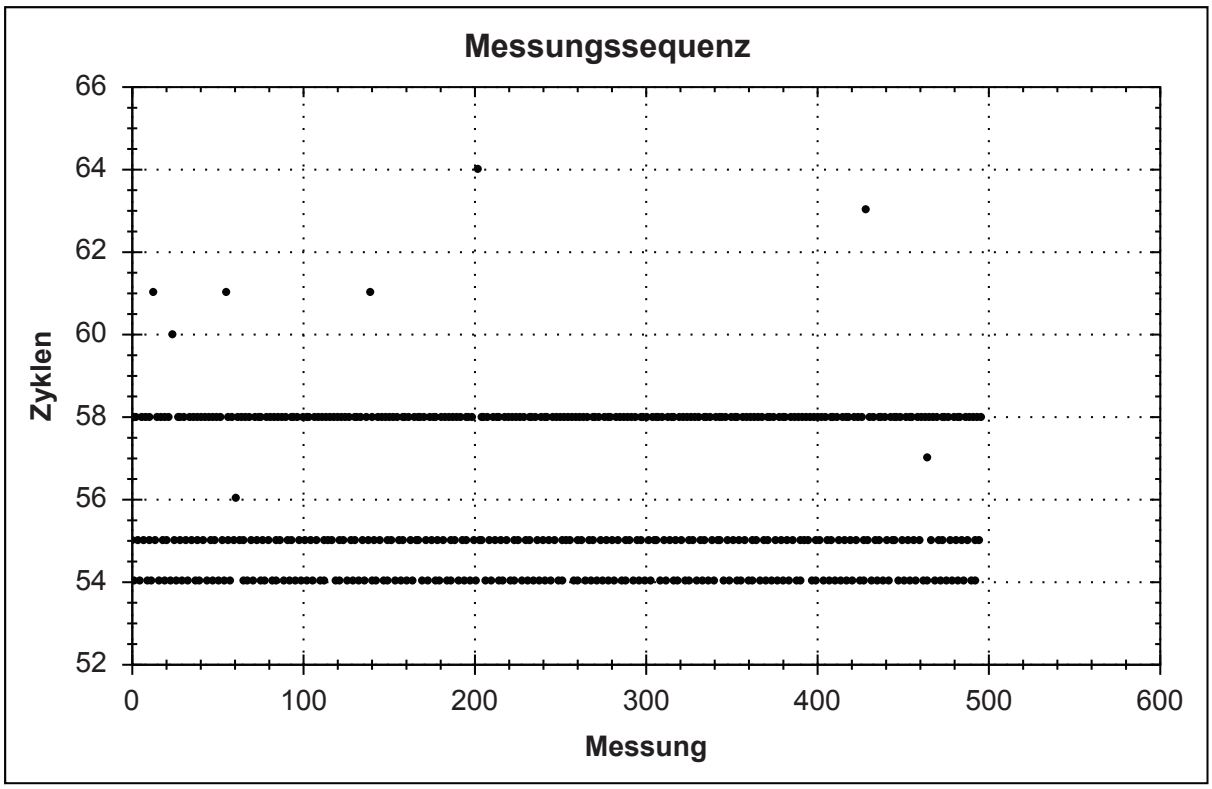


Abb. 5.5: Verlaufskurve Laufzeitmessung (Laufzeitprofil)

5.1.4 Struktursuche

In diesem Verfahrensschritt erfolgt das Auffinden von Strukturen. Das hier beschriebene Verfahren geht über die derzeitig verfügbaren Methoden der dynamischen Laufzeitanalyse hinaus, indem vollständige Programmpfadinformationen mit Laufzeitinformationen kombiniert werden. Wie in Kapitel 3 gezeigt wurde, werden in bisherigen dynamischen Prüfmethoden die Laufzeitinformationen nicht mit den Programmpfadinformationen verknüpft, die Prüfung erfolgt meist auf Basis der Ergebnisse der Wahrscheinlichkeitsrechnung. Bei statischen und hybriden Prüfmethoden ist die Kombination von Laufzeitinformationen und Programmablaufinformationen wesentlicher Bestandteil der Berechnung von Laufzeitgrenzen, dies erfolgt jedoch auf Basis von Teilpfadinformationen z. B. auf Grundblockebene. Bei bisherigen Prüfmethoden gehen die Zusammenhänge zwischen Testdaten, Laufzeiten und vollständigen Programmpfadinformationen verloren bzw. stehen nicht zur Verfügung.

Einen neuen Weg zeigt das hier beschriebene Verfahren auf. Auch in diesem Analyseschritt wird keine Stochastik, also auf Wahrscheinlichkeitsrechnung basierende Verfahren, benutzt, aber einige der verwendeten Methoden sind durchaus von der Statistik beeinflusst. Über die Darstellung von Daten (s. Abb. 5.5) hinaus wird hier die Datenanalyse zur Suche nach Strukturen erweitert. Das Konzept zur Datenfusion und Struktursuche wird nachfolgend beschrieben.

Datenfusion: Der Bezug der Eingabedatenabhängigkeit der Prozesslaufzeit (mit dem Merkmalspaar $[d_{n,i};\Delta t^{j}_{\mathrm{Pr}\,E,i}]$) und der Eingabedatenabhängigkeit des Programmpfades (mit dem Merkmalspaar $[d_{n,i};p_{n,i}]$) wird über die Eingabe $d_{n,i}$ hergestellt. Die Verknüpfung der Entitäten Pfad und Messung erfolgt über die Beziehung Messung-Pfad (s. Abb. 5.6). Um die Beziehung zwischen Messung und Pfad herzustellen, werden zu den Eingaben der Messung die gleichen Eingaben zu einem Pfad gesucht. Dieser Vorgang wird als Synchronisierung bezeichnet und schließt die Datenfusion ab.

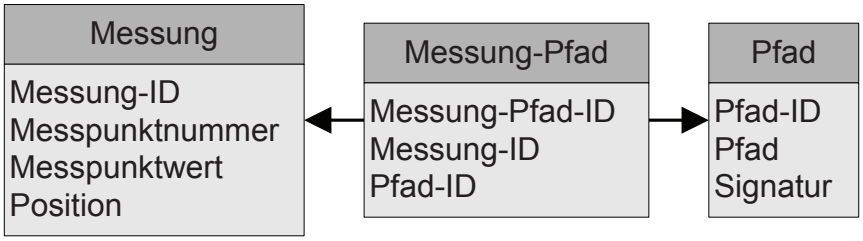


Abb. 5.6: Laufzeit zu Pfadbeziehung

Die Zusammenhänge sind am Beispielprogramm aus Kapitel 4 in Tabelle 5.3 dargestellt. In Abb. 5.7 sind die Laufzeiten den jeweiligen Pfaden durch entsprechende farbliche Differenzierung zugeordnet. Der Gewinn an Aussagekraft resultiert hier einzig aus der Zuordnung von Pfadinformationen zu den entsprechenden Laufzeiten. Erst so sind Architektur und Programmpfadabhängigkeiten zu beurteilen.

Tabelle 5.3: Entitäten und Beziehung von Messung-Pfad

Messung			
ID	MP-Nr.	MP-Wert	Position
1	t_1	4072634	0
2	t_2	4072689	0
3	t_1	4811128	1
4	t_2	4811186	1
5	t_1	5568742	2
6	t_2	5568796	2
7	t_1	6323640	3
8	t_2	6323698	3
*	*	*	*

Messung-Pfad		
ID	Messung-ID	Pfad-ID
1	1	1
2	2	1
3	3	2
4	4	2
5	5	3
6	6	3
7	7	2
8	8	2
*	*	*

Pfad		
ID	Pfad	Signatur
1	z_1, z_2	t_1, z_1, z_2, t_2
2	z_1, z_3, z_4	t_1, z_1, z_3, z_4, t_2
3	z_1, z_3, z_5	t_1, z_1, z_3, z_5, t_2

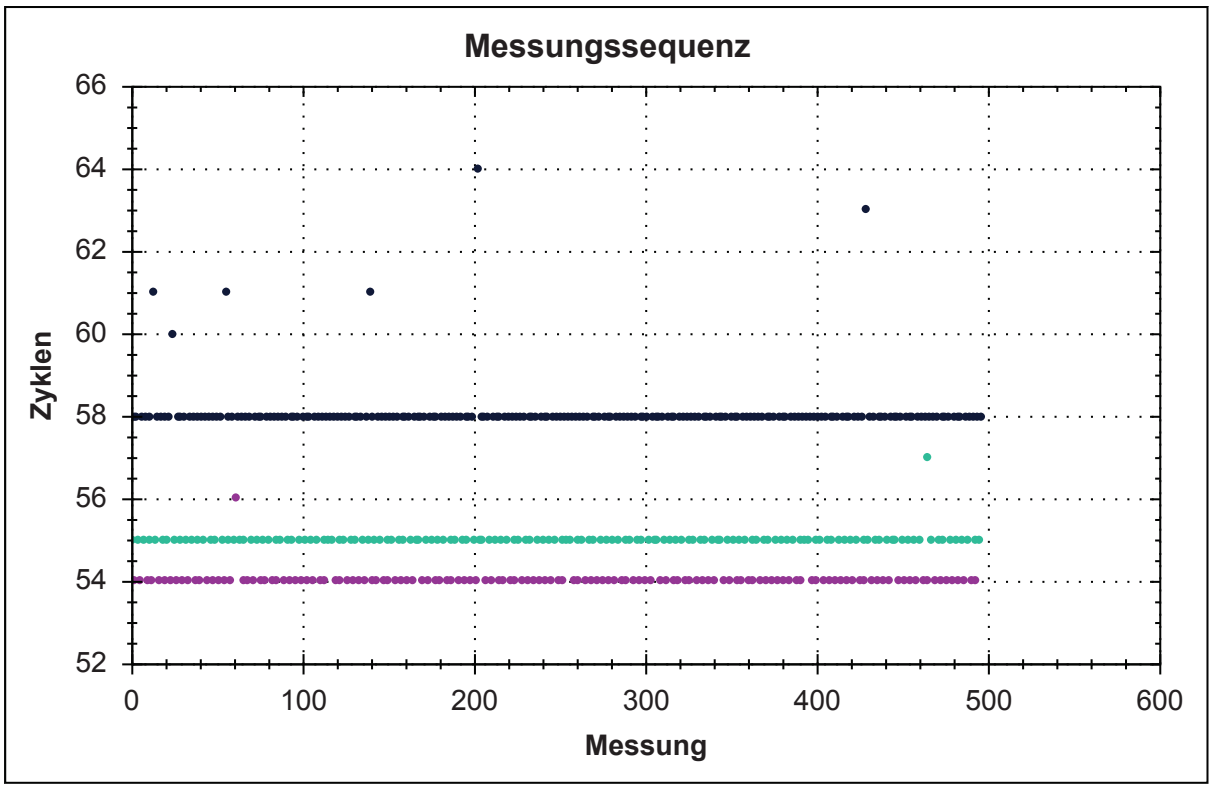


Abb. 5.7: Laufzeitprofil mit Pfaddarstellung: Pfad-ID 1 Pfad-ID 2 Pfad-ID 3

Struktursuche: Durch Kombination der Daten können Pfadabhängigkeiten bestimmt und nach Besonderheiten in den Daten, wie pfadabhängige WCET, gesucht werden. Dies ermöglicht Aussagen über die pfadabhängigen Laufzeitunterschiede innerhalb eines Prozesses und auf Programmpfadebene eine Bewertung von Architektureinflüssen. Die Bewertung von pfadabhängigen Laufzeitmessdaten soll anhand von Experimenten im nächsten Abschnitt beschrieben werden.

5.2 Experimente zur programmpfadorientierten Datenanalyse

Das Ziel der hier dargestellten Fallstudie ist es, die Wirksamkeit des temporalen Prüfkonzeptes und deren Implementierung in Techniken und Werkzeuge zu demonstrieren. Im folgenden Abschnitt werden anhand der exemplarischen Software-Funktion PR_TimMdl die Ergebnisse der Laufzeitmessungen und pfadabhängigen Datenanalyse vorgestellt. Die Messungen erfolgten nach dem unter Abschn. 4.4 beschriebenen Messverfahren. Die Analyse beschränkt sich auf die Auswertung der Messungen in Hinblick auf WCET und BCET der gemessenen Pfade sowie auf die Darstellung der Laufzeiten.

Eine Interpretation der Laufzeiten unter Betrachtung der Hardwareeinflüsse durch Pipeline-, Cache- und Multitasking-Effekte erfolgt im Kapitel 6. Das in Abb. 5.8 dargestellte ASCET-Modell der Software-Funktion PR_TimMdl wird im Folgenden bzgl. der Software-Eigenschaften beschrieben. Der Kontrollflussgraph ist im Anhang A.3 dargestellt.

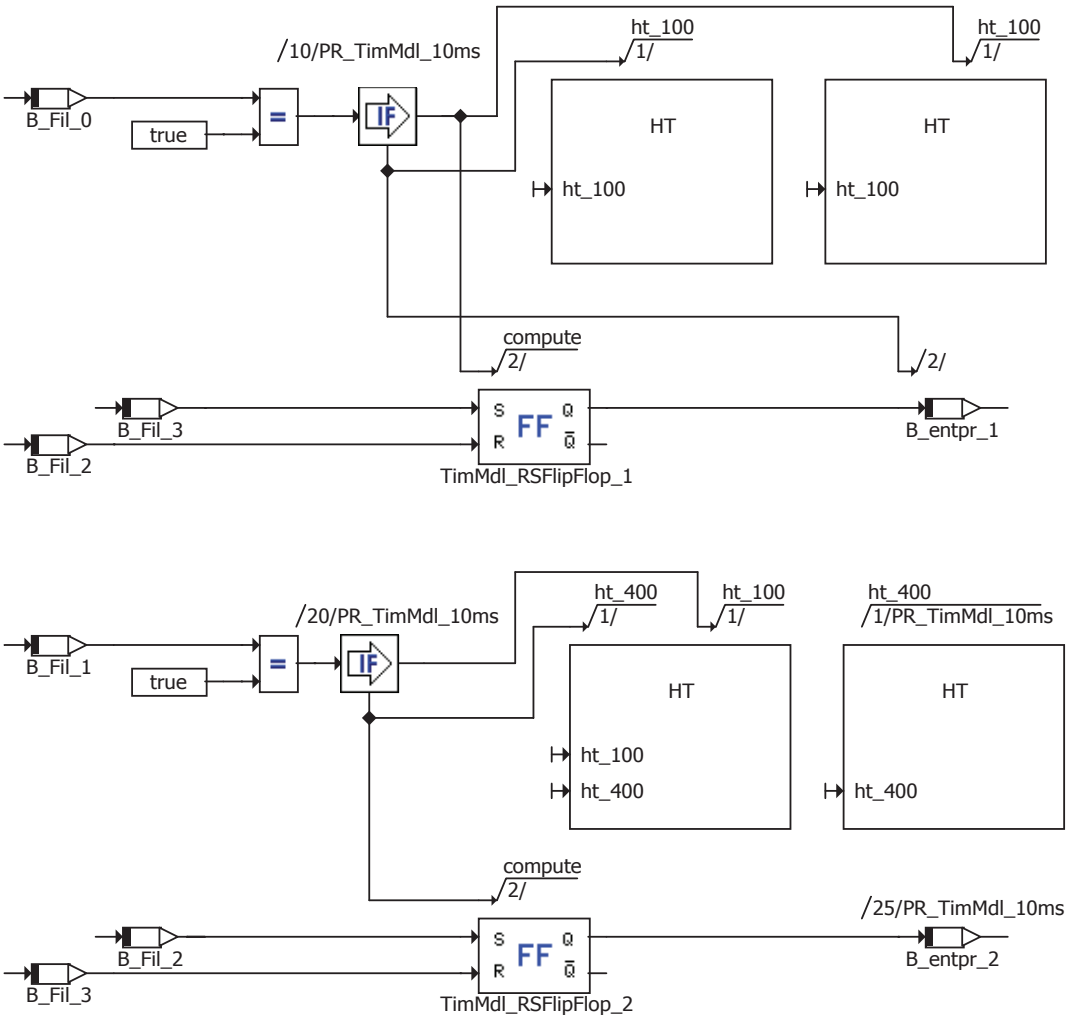


Abb. 5.8: ASCET-Modell der Software-Funktion PR_TimMdl

5.2.1 Metriken, Programmzweig- und Schnittstellenanalyse

Die Software-Funktion besteht aus dem Prozess PR_TimMdl_10ms() (kurz _10ms()). In der Funktion _10ms() werden logische Operationen durchgeführt und Unterfunktionen aufgerufen.

Metriken: Die hier für Steuergeräte-Software relevanten Metriken sollen lediglich einen Überblick über die Testbarkeit der Software geben. Wichtige Elemente sind dabei:

- Anzahl der aufrufenden Funktionen einer Funktion NBCALLING (≤10),
- Anzahl der Pfade einer Funktion PATH (≤1000),
- Anzahl aufgerufener Funktionen einer Funktion DC_CALLING (≤7) und
- maximale Schachtelungstiefe der Kontrollstrukturen einer Funktion LEVEL (≤4).

Die Funktion PR_TimMdl erfüllt die vorgegebenen Software-Metriken des Qualitätsmodells (s. Abschn. 4.2.1), die Metriken sind im Anhang A.3 dargestellt.

Programmzweiganalyse: Die statische Codeanalyse stellt eine Auflistung der Programmzweige zur Verfügung, um eine Zweigabdeckung feststellen zu können. Die Aufzählung der Zweiginstrumentierungen basiert auf dem im Anhang A.3 abgebildeten Kotrollfluss.

PR_TimMdl: /1/2/3/4/5/1001/1002/1003/1004/1005

Schnittstellenanalyse: Generell werden alle externen Schnittstellensignale der Software-Funktion aufgenommen, hierzu zählen auch Zustandsgrößen in gedächtnisbehafteten Systemen. In Tabelle 5.4 findet sich die Zusammenstellung der Variablen/Signale, welche zur Funktion PR_TimMdl gehören. Hierüber wird der Bezug zur Eingabedatenabhängigkeit der Prozesslaufzeit und der Eingabedatenabhängigkeit des Programmpfades hergestellt (s. Abschn. 5.1.4).

Tabelle 5.4 Eingangs- und Ausgangssignale der Funktion

Variable/Signale	Datentyp	Wertebereich	Beschreibung
B_Fil_0	Bit	true/false	Eingang
B_Fil_1	Bit	true/false	Eingang
B_Fil_2	Bit	true/false	Eingang
B_Fil_3	Bit	true/false	Eingang
B_entpr_1	Bit	true/false	Ausgang
B_entpr_1	Bit	true/false	Ausgang

5.2.2 Pfadverfolgung und Laufzeitmessung

Instrumentierung: Die Instrumentierung des Programmcodes erfolgt automatisch mit der Generierung des Quellcodes aus dem Modell. Das Ergebnis dieser Instrumentierung ist der Testcode (unter Zweigkontrolle) für den Software-Modultest und der Messcode (unter Laufzeitkontrolle) für den temporalen Regressionstest.

Erzeugung des Test- und Messobjektes: Im nächsten Schritt wird aus dem Testcode und dem Messcode ein ausführbarer Objektcode erzeugt und in das Motorsteuergerät geladen.

Pfadverfolgung: Die Pfadverfolgung zur Software-Funktion wurde am Messplatz mit manueller Stimulierung der Eingangsgrößen vorgenommen. Manuelle Stimulierung bedeutet, dass ein Software-Modultest unter der Vorgabe von Testfällen erfolgt, wobei alle ausführbaren Programmpfade einmal erreicht wurden. Das Ergebnis der Pfadanalyse bzw. die Entität Pfad ist in Tabelle 5.5 dargestellt.

Tabelle 5.5: ausgeführte Programmpfade der Software-Funktion PR_TimMdl

Pfad-ID[a]	Signatur
1	/t1/z1/z3/z5/z1001/z1003/z1005/t2
2	/t1/z1/z2/z1001/z1003/z1005/z5/z1001/z1003/z1005/t2
3	/t1/z1/z3/z4/t2
4	/t1/z1/z2/z1001/z1003/z1005/z4/t2
5	/t1/z1/z3/z5/z1001/z1003/z1004/t2
6	/t1/z1/z2/z1001/z1002/z5/z1001/z1003/z1004/t2
7	/t1/z1/z2/z1001/z1002/z4/t2
8	/t1/z1/z3/z5/z1001/z1002/t2
9	/t1/z1/z2/z1001/z1003/z1004/z5/z1001/z1002/t2
10	/t1/z1/z2/z1001/z1003/z1004/z4/t2
11	/t1/z1/z2/z1001/z1002/z5/z1001/z1002/t2

a – Pfad-ID eindeutige Nummerierung jedes unterschiedlichen Programmpfades

Laufzeitmessung: Die Eingangssignale der Pfadverfolgung wurden entsprechend Abschnitt temporaler Regressionstest aufgezeichnet und zur Stimulierung der ersten Laufzeitmessung auf dem Motorsteuergerät verwendet. Die in den Diagrammen und Tabellen angegebenen WCETs, BCETs und Durchschnitte sind die gemessenen Zyklen c^j und entsprechen den CPU-Zyklen. Eine Umrechnung in die Prozesslaufzeit kann nach Gl. 4.2: $\Delta t^j_{\mathrm{Pr}\,E} = c^j * t_{Timerzyklen}$ mit $t_{Timerzyklen} = 1/75\,\mathrm{MHz}$ erfolgen. Die angegebenen Durchschnitte entsprechen dem arithmetischen Mittel aller Messungen des jeweiligen Pfades.

5.2.3 Testdokumentation

Wurden die Messungen durchgeführt und die Daten synchronisiert, kann die Datenanalyse erfolgen. Dabei wird die Testdokumentation erstellt, und die Messdaten werden für die Laufzeitanalyse ausgegeben. Die Laufzeiten der Programmpfade können dann in den Diagrammen der Testdokumentation analysiert werden. Die Synchronisation der Pfadverfolgung und der Laufzeitmessung erfolgte über die Eingangs- und Ausgangssignale der Funktion.

Laufzeitmessung A1

Bei der ersten Messung (Pfadabdeckung) werden ca. 310 Messungen im 10 ms Task (ca. 3,1 Sekunden Messdauer) betrachtet. Die Testfälle werden bis zum Ende der Messung zyklisch wiederholt. Abbildung 5.9 zeigt das Ergebnis der Messung. Durch Übernahme der Testfälle aus dem Software-Modultest wurden 11 Pfade der Funktion stimuliert, wobei alle ausführbaren Programmpfade einmal erreicht wurden.

Laufzeitmessung A2

Die zweite Messung wurde im Rahmen der funktionalen Tests in der Musterphase am Fahrzeug durchgeführt. Dabei wurden die Laufzeiten und das ablaufende Fahrprofil aufgezeichnet. Das bedeutet, dass die Eingangs- und Ausgangssignale protokolliert werden, wie sich diese in Abhängigkeit des Fahrprofils (Testeingaben durch den Fahrer) bei jedem Aufruf der Funktion verändern. Wie bereits im Kapitel 4 beschrieben, verfügt die Testumgebung über die Möglichkeit, mit aufgezeichneten Fahrprofilen über die Steuergeräteschnittstellen die Software-Funktionen zu stimulieren. Mit dem in Abschn. 4.3 vorgestellten temporalen Regressionstestwerkzeug ist eine automatische Stimulierung der Eingangsgrößen möglich. Die funktionalen Testfälle können hiermit nachträglich durch Pfadverfolgung oder Laufzeitmessung überprüft werden. In Abb. 5.10 ist die automatische Stimulation des _10ms() Prozesses durch ein Fahrprofil dargestellt.

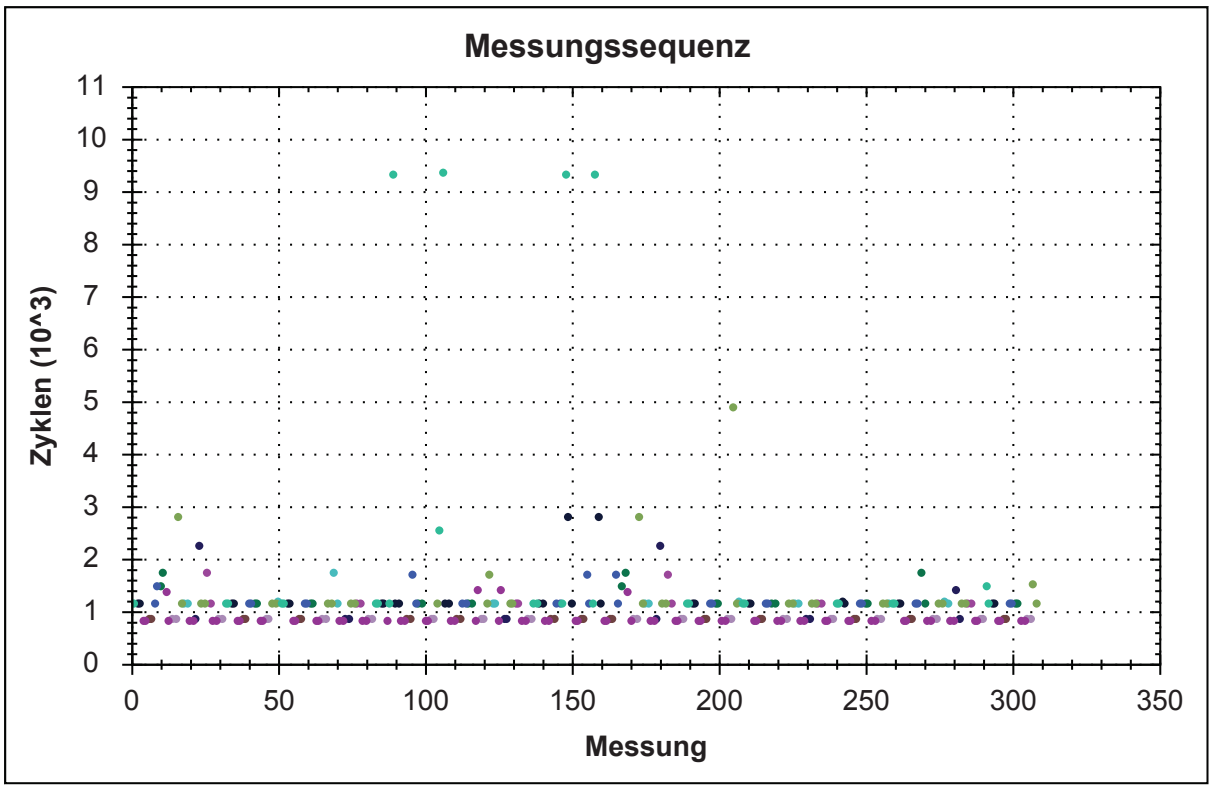


Abb. 5.9: Laufzeitmessung A1 der Funktion PR_TimMdl_10ms() - Pfadabdeckung

Für die Funktion _10ms() ergeben sich für die gemessenen Pfade folgende Laufzeiten (s. Tabelle 5.6).

Tabelle 5.6: Laufzeitmessung A1 der Funktion PR_TimMdl_10ms()

ID	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
1		9338	1143	2372
2		2808	1148	1276
3		1391	820	851
4		835	835	835
5		1709	1146	1223
6		1732	1148	1268
7		837	836	836
8		4866	1144	1384
9		1741	1162	1201
10		2257	838	1121
11		1717	1146	1251

a Darstellung in Abb. 5.9, b Timerzyklen *c*

Aus den Laufzeiten der einzelnen Pfade werden folgende WCET und BCET der Funktion _10ms() (s. Tabelle 5.7) bestimmt.

Tabelle 5.7: gemessene WCET/BCET bei A1 der Funktion PR_TimMdl_10ms()

	Pfad	Farbe[a]	Zyklen[b]
WCET	/1/3/5/1001/1003/1005		9338
BCET	/1/3/4		820
Durchschnitt	aller Pfade		1238

a und b siehe Tabelle 5.6

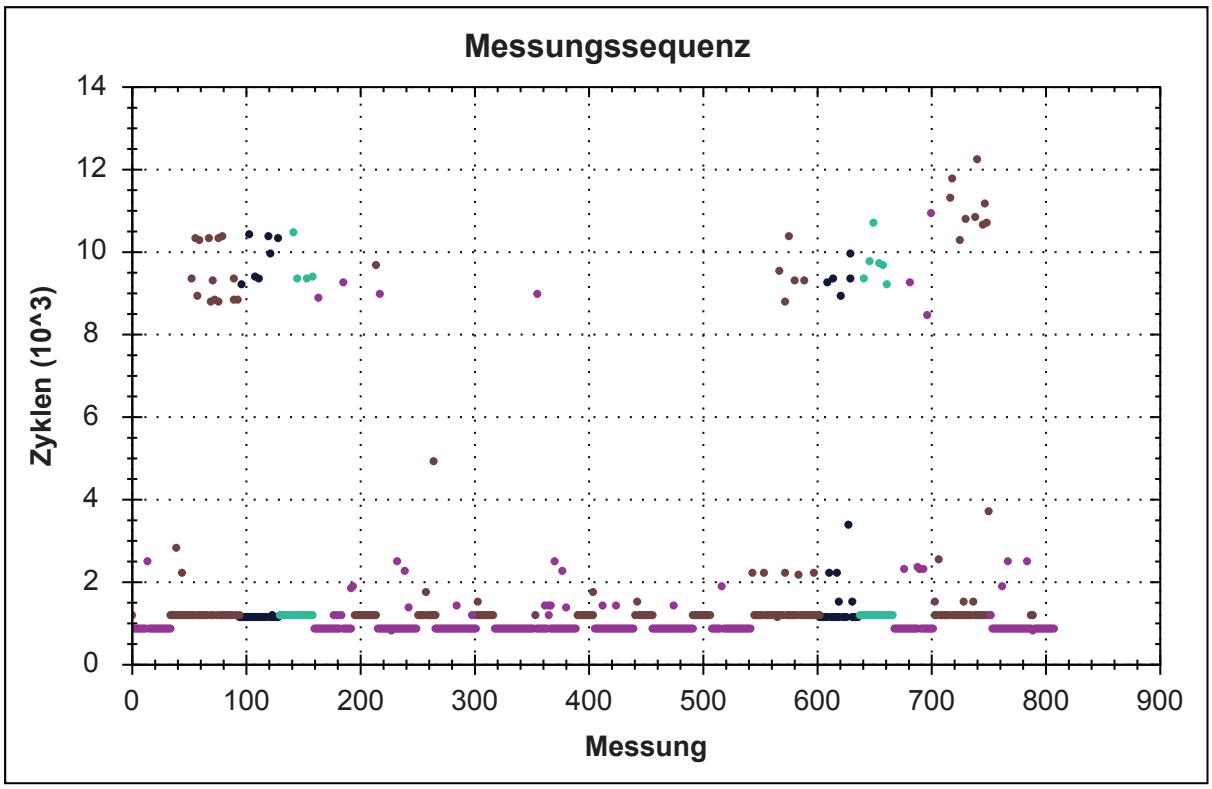


Abb. 5.10: Laufzeitmessung A2 der Funktion PR_TimMdl_10ms() - Zweigabdeckung

Für die Funktion _10ms() ergeben sich für die gemessenen Pfade folgende Laufzeiten (s. Tabelle 5.8).

Tabelle 5.8: Laufzeitmessung A2 der Funktion PR_TimMdl_10ms()

ID	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
2		10697	1158	2579
5		10391	1146	2674
7		10918	823	1052
9		12231	1148	2161

a Darstellung in Abb. 5.10, b Timerzyklen *c*

Aus den Laufzeiten der einzelnen Pfade werden folgende WCET und BCET der Funktion _10ms() (s. Tabelle 5.9) bestimmt.

Tabelle 5.9: gemessene WCET/BCET bei A2 der Funktion PR_TimMdl_10ms()

	Pfad	Farbe[a]	Zyklen[b]
WCET	/1/2/1001/1003/1004/5/1001/1002		12231
BCET	/1/2/1001/1002/4		823
Durchschnitt	aller Pfade		2116

a und b siehe Tabelle 5.8

Die Messung A2 mit automatischer Stimulierung durch ein Fahrprofil dient hier der Demonstration, um zu zeigen, dass gezielte Untersuchungen von Fahrsituationen möglich sind. Hier sollte beachtet werden, dass alle gezeigten Messungen auf einem seriennahen Motorsteuergerät (C Muster [25]) mit einem vollfunktionsfähigen Testprogrammstand (s. Abschn. 2.5.2 Testintegration) durchgeführt wurden. Das heißt, alle Messungen unterliegen den typischen Echtzeiteinflüssen (s. Kapitel 2) wie Cache- und Multitasking-Effekte.

In den folgenden Abschnitten soll gezeigt werden, wie die zuvor erzeugten Messungssequenzen (Diagramme) effizient repräsentiert und geprüft werden können. Dabei wird auch auf die Beurteilung der Vollständigkeit des temporalen Tests über die Pfadabdeckung des Programms sowie den Stichprobenumfang und die Bewertung der temporalen Bedingungen, unter denen der Test erfolgt, eingegangen.

5.2.4 pfadabhängige Laufzeitanalyse

Die explorative Datenanalyse, wie sie im Folgenden durchgeführt wird, ist eine Weiterführung und Vereinfachung der deskriptiven Datenanalyse durch das Messsequenzdiagramm und wird als pfadabhängige Laufzeitanalyse betrachtet. Rückschlüsse über die Erhebung hinaus sind in diesem Verfahrensschritt nicht durch Wahrscheinlichkeitsaussagen formalisiert, es ergeben sich dennoch deutliche Hinweise bezüglich Pfad- und Architekturabhängigkeiten. Hierfür erfolgt eine Betrachtung der Pfadlaufzeiten als WCET/BCET-Diagramm. Im Diagramm WCET/BCET werden alle Pfade mit den zugehörigen WCET und BCET dargestellt. Auf der Abszisse befindet sich die ID des Pfades, wie sie in der Tabelle 5.5 benutzt wird. An der Ordinate werden die BCET und die WCET in Zyklen aufgetragen. Mit diesem Schritt ist das Problem der Abbildung von Laufzeiten auf die Programmpfade auch für komplexe Hardwarearchitekturen gelöst. Abbildung 5.11 zeigt das WCET/BCET-Diagramm der Laufzeitmessung A1 und Abb. 5.12 der Laufzeitmessung A2.

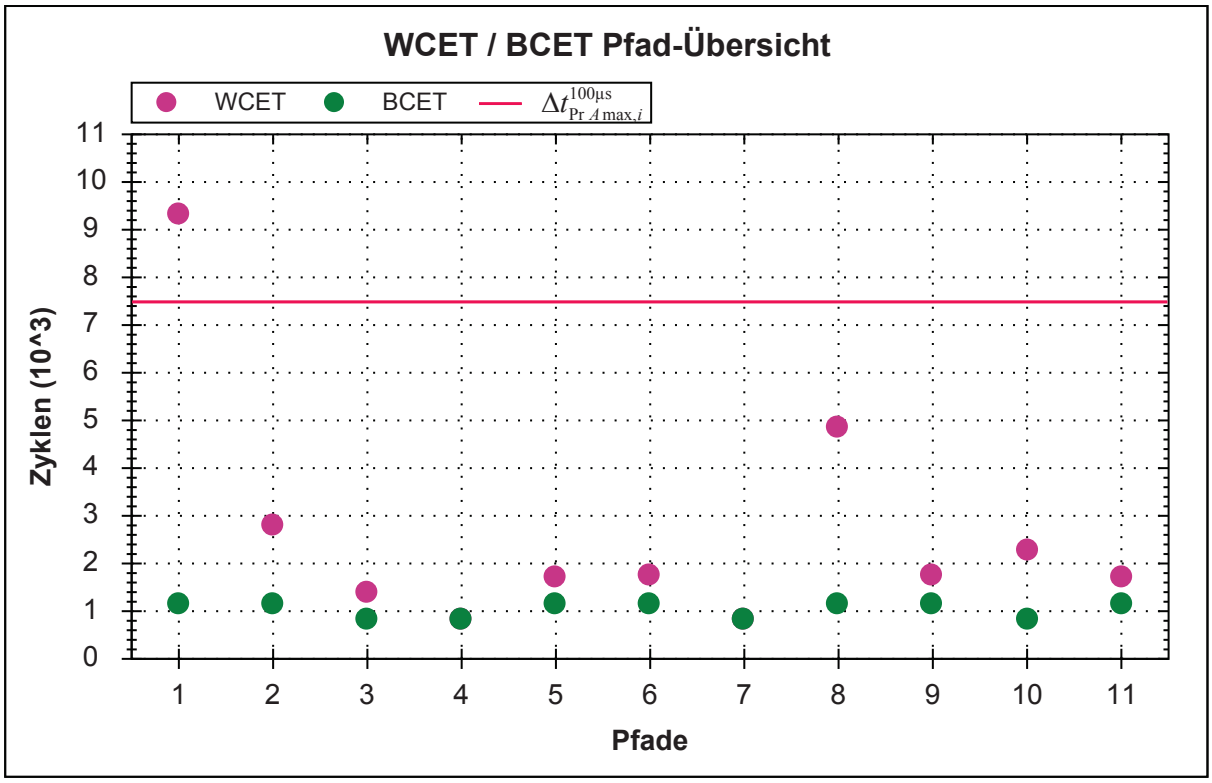


Abb. 5.11: WCET/BCET-Diagramm der Laufzeitmessung A1 - Pfadabdeckung

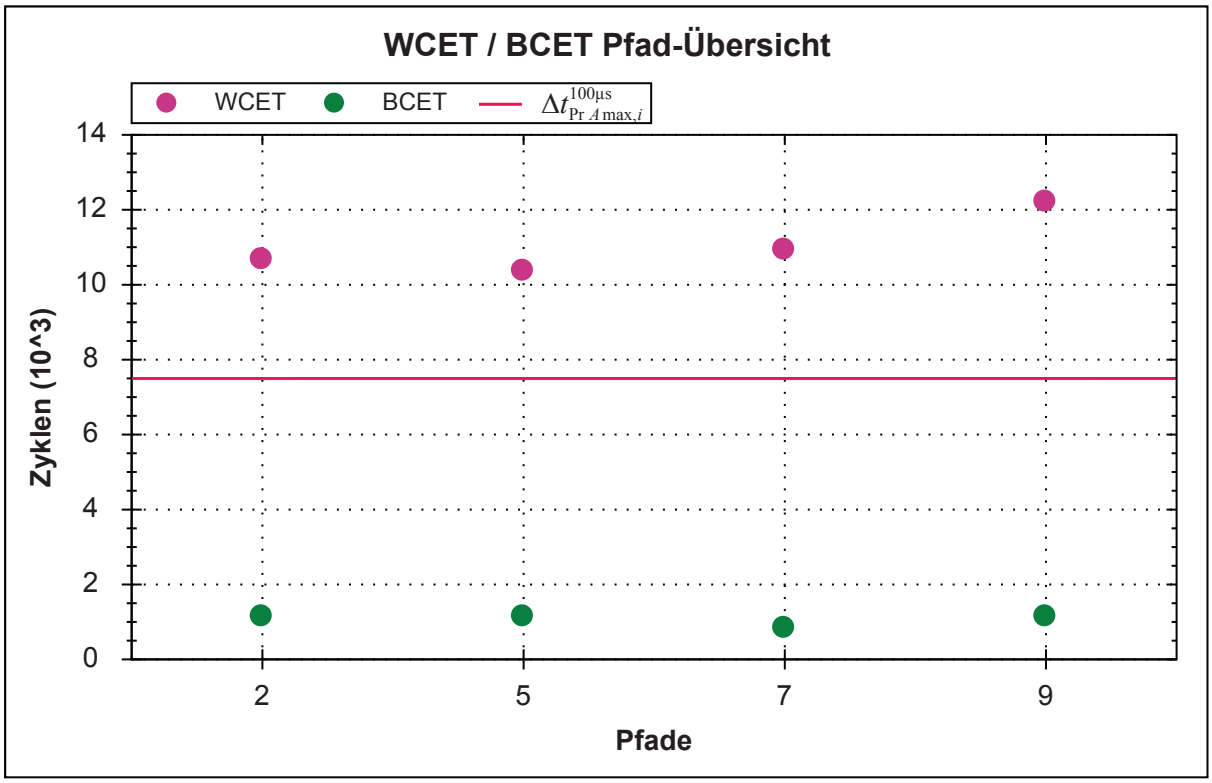


Abb. 5.12: WCET/BCET-Diagramm der Laufzeitmessung A2 - Zweigabdeckung

5.2.5 Laufzeitprüfung

Beim Vergleichen der aufgezeichneten Ist-Laufzeiten $\Delta t^{j}_{\Pr E,i}$ mit der maximal zulässigen Prozesslaufzeit $\Delta t^{100\mu s}_{\Pr A\max}$ (dem spezifizierten Prozess-Container) des Programms ist festzustellen, dass die gemessenen WCET bei Messung A1 mit 9338 Zyklen und bei Messung A2 mit 12231 Zyklen über den geforderten 7500 Zyklen liegen. Die Prüfung, ob das Programm die nach der Spezifikation geforderten Laufzeiteigenschaften erfüllt, muss hier zum Ergebnis kommen, dass die Laufzeitrestriktion verfehlt wird.

Zur Beurteilung der Ergebnisse aus den Messungen A1 und A2 soll die Vollständigkeit des temporalen Tests, wie in Abschnitt 4.2 beschrieben, aus der Pfadabdeckung des Programms unter definierten temporalen Bedingungen berücksichtigt werden. Eine vollständige Pfadabdeckung (wie in Laufzeitmessung A1) ist meist nur mit hohem Aufwand zu erreichen, da dazu die mitunter sehr hohe Anzahl an Programmpfaden stimuliert und gemessen werden muss. Die Vollständigkeit des Tests durch die Pfadabdeckung des Programms zu beurteilen, besitzt daher keine praktische Bedeutung. Die erreichte vollständige Zweigabdeckung (wie in Laufzeitmessung A2) wird in der Praxis häufig gefordert und kann als Maß des strukturorientierten Tests im Software-Modultest genügen. Die Laufzeiteigenschaften des Prozesses - unter den gegebenen Testbedingungen - können jedoch nur beschrieben werden, wenn bei vollständiger Zweigabdeckung auch alle Pfade abgedeckt werden. Aus den Ergebnissen der Laufzeitmessung A2 mit Zweigabdeckung kann daher keine Aussage über die Einhaltung oder Verletzung von Laufzeitrestriktion für die nicht gemessenen Pfade getroffen werden. Dies wird auch aus den unterschiedlichen Ergebnissen für die ermittelten WCET und BCET der Funktion _10ms() und deren Pfade beim Vergleich der Messungen A1 und A2 sichtbar (s. Tabelle 5.7 und 5.9).

Neben der Vollständigkeit der Ergebnisse, die durch die Qualitätssicherung (s. Abschn. 1.1.2) gefordert wird, muss auch die Verlässlichkeit der Ergebnisse sichergestellt sein. Um die Er-

gebnisse der Messungen zu verifizieren, ist eine Bestimmung der Messgenauigkeit des Verfahrens notwendig. Durch das Messen auf der Zielarchitektur werden Pipeline- und Cache-Effekte mit in die Messung aufgenommen. Für die sichere Bestimmung der Prozesslaufzeit sind diese Effekte zu berücksichtigen. Eine genaue Analyse ist jedoch nur durch eine Untersuchung der zum Einsatz kommenden Hardware möglich.

In der hier betrachteten Multitasking-Umgebung können die gemessenen Laufzeiten durch nicht beherrschbare, nicht deterministische Einflüsse überlagert sein, die auf die Unterbrechungen des Programmablaufs durch Tasks mit höherer Priorität zurückzuführen sind. Diese Unterbrechungen sind durch breite Laufzeitintervalle[1] in den WCET/BCET-Diagrammen (s. Abb. 5.11 und 5.12) erkennbar. Die gemessenen Laufzeiten beinhalten hier die Laufzeit der unterbrechenden Tasks und entsprechen damit nicht der Prozesslaufzeit, daher sind die gemessenen Laufzeiten durch die Unterbrechung der Prozessausführung als Fehlmessung zu interpretieren. Im Kapitel 6 wird eine Methode vorgeschlagen, um Fehlmessungen zu erkennen und damit aus der Datenanalyse herauszuhalten.

Fazit: Die abschließende Bewertung, ob eine korrekte Umsetzung der temporalen Anforderungen erfolgte, kann erst nach der Bewertung der Randbedingungen des Tests erfolgen.

5.3 Grenzen der programmpfadorientierten Datenanalyse

Um das temporale Verhalten von Software des Motorsteuergerätes zu testen, wurde ein Messprozess entwickelt, der die Prozesslaufzeiten auf der Zielarchitektur messtechnisch ermittelt. Im Rahmen dieser Arbeit wurde dazu als Teil der Prüfstrategie zur Auswertung der Messdaten des temporalen Tests ein Datenanalyseverfahren implementiert. Die Ergebnisse der Prüfstrategie werden durch die Messdatenorganisation strukturiert und einfach dokumentiert. Dabei entsteht eine durchgängige und jederzeit reproduzierbare Testdokumentation. Das Auffinden der gemessenen pfadabhängigen WCET und BCET wird in der Implementierung durch den Einsatz einer Datenbank ermöglicht, die eine effiziente und strukturierte Analyse der Messdaten zulässt.

Die Methode der programmpfadorientierten Datenanalyse lässt lediglich die Überprüfung von Testfällen auf den tatsächlich durchlaufenen Programmpfad und dessen Laufzeit zu. Weiterhin können bei Versuchsfahrten oder am Prüfstand aufgenommene Daten (Fahrprofile) auf ihr Laufzeitverhalten untersucht werden, vollständige Ergebnisse können nicht geliefert werden. Der Nachteil des Verfahrens ist, dass für die getestete Software-Funktion nicht sichergestellt ist, dass ein vollständiges Ergebnis vorliegt, wenn nicht alle unterschiedlichen vollständigen Pfade unter allen möglichen Randbedingungen getestet werden. Typischerweise sollte die pfadabhängige Laufzeitbestimmung dann eingesetzt werden, wenn wenige Programmpfade existieren oder wenn Testfälle gezielt bestimmte Programmsituationen überprüfen sollen. Das Verfahren liefert jedoch einen ersten Anhaltspunkt für die WCET vor der Software-Integration. Eine erneute Messung ist aber unumgänglich, wie Messung A2 zeigt.

Im folgenden Kapitel wird die Bewertung von Laufzeitmessergebnissen vorgenommen, die auf Basis des beschriebenen Prüfkonzeptes ermittelt werden. Dafür werden die Zusammenhänge zwischen dem am Messobjekt gemessenen Wert und dem wahren Wert betrachtet. Im Kapitel 7 wird eine Ergänzung der programmpfadorientierten Datenanalyse um eine funktionspfadorientierte Datenanalyse vorgeschlagen, um verlässliche und formal vollständige Ergebnisse angeben zu können.

[1] Laufzeitdifferenz zwischen der WCET und der BCET eines Programmpfades, der unter Wiederholbedingungen betrachtet wird

6 Bewertung der Laufzeitmessergebnisse

Reale Laufzeitmessungen, die auf einer Instrumentierung des Programmcodes basieren, sind immer auch mit einer Messabweichung verbunden. Zur Bewertung der Qualität der Laufzeitmessung muss diese Abweichung qualifiziert werden. Teil der Auswertung der Laufzeitmessung ist damit, eine Abschätzung der Brauchbarkeit der Laufzeitmessung auf Basis der auftretenden Messabweichung herzustellen. Hierfür müssen die Ursachen der Messabweichung [43] bestimmt sein, im Wesentlichen lassen sich diese auf:

- Rückwirkungen der Messeinrichtung auf das Messobjekt,
- Unvollkommenheit des Messgerätes oder der Messwertverarbeitung und
- Umwelteinflüsse auf die Messeinrichtung oder Störungen, die dem Messsignal überlagert sind, zurückführen.

Wie in Kapitel 2 eingeführt wurde, kann die Messabweichung e in systematische Abweichungen e_s und zufällige Abweichungen e_r unterschieden werden. Die systematische Messabweichung hat während der Messung einen konstanten Betrag mit bestimmten Vorzeichen und führt immer zu einer gleichen, zeitlich konstanten Differenz des Messwertes vom wahren Wert. Die systematische Messabweichung ist unter Wiederholbedingungen nicht erkennbar. Die bekannte systematische Messabweichung, d. h. Betrag und Vorzeichen sind bekannt, kann korrigiert werden. Die unbekannte systematische Messabweichung, d. h. der Teil der systematischen Messabweichung die vermutet werden, deren Betrag und/oder Vorzeichen nicht eindeutig angegeben werden kann, kann nicht korrigiert werden. Die zufällige Messabweichung wird bestimmt durch nicht beherrschbare, nicht determinierte Einflüsse während der Messung, diese können im Voraus nicht bestimmt werden. Unter Wiederholbedingungen führen sie zu einer Streuung der Messwerte, d. h. zu einem unsicheren Messergebnis.

6.1 Umgang mit Messgeräteabweichung

Zunächst muss unterschieden werden zwischen der Abweichung des Messergebnisses vom wahren Wert der Messgröße und der in ihr enthaltenen Messabweichung durch das verwendete Messwerkzeug. Die Messabweichung, die vom Messwerkzeug hervorgerufen wird, wird als Messgeräteabweichung e_M bezeichnet und kann in systematische Abweichungen e_{Ms} und zufällige Abweichungen e_{Mr} unterschieden werden:

$$e_M = e_{Ms} + e_{Mr} \tag{6.1}$$

Mit Gl. 6.1 kann die Gl. 2.12, bezogen auf den Laufzeitmesswert t unter Vernachlässigung der nicht von Messgeräten verursachten Messabweichung, wie folgt ergänzt werden:

$$t = t_w + e_M = t_w + e_{Ms} + e_{Mr} \tag{6.2}$$

In der Messtechnik [43] wird an der Stelle des nicht zugänglichen wahren Wertes t_w der Messgröße der richtige Wert der Messgröße t_r verwendet: $t = t_r + e_M$. Daraus folgt für die Messgeräteabweichung:

$$e_M = t - t_r = \Delta t \tag{6.3}$$

Anstelle der Angabe der absoluten Messgeräteabweichung e_M ist die Angabe der relativen Abweichung eines Messgerätes üblich. Die Bezugsgröße ist dann oft nicht der wahre Wert oder der richtige Wert eines Messergebnisses, sondern ein messgerätetypischer Wert wie der Endwert des gewählten Messbereiches. Damit sind drei Messbereiche für eine Motorsteuerung nach Kapitel 2 relevant, diese lassen sich nach der Größe der Prozess-Container $\Delta t_{\mathrm{Pr}\,A\max}$ in 10µs, 100µs und 200µs einteilen. Der Bezug für die relative Abweichung eines Laufzeitmessgerätes lautet damit:

$$e_{Mrel} = \frac{e_M}{\Delta t_{\mathrm{Pr}\,A\max}} \tag{6.4}$$

Mit $\Delta t_{\mathrm{Pr}\,A\max}$ wird der gewählte Bezugswert auf den Endwert des Messbereiches festgelegt. Eine Abschätzung der Messgeräteabweichung ist nur für den systematischen Anteil möglich, wenn der richtige Wert der Messgröße auf geeignete Weise zugänglich wurde. Der richtige Wert t_r der Messgröße kann durch entsprechend aufwendige Verfahren ermittelt werden. Von diesem Wert kann dann angenommen werden, dass er eine vernachlässigbare Abweichung vom wahren Wert der Messgröße besitzt. Zwei dieser aufwendigen Verfahren werden nachfolgend betrachtet, der erste Ansatz beinhaltet eine theoretische Betrachtung und der zweite Ansatz praktische Messaktivitäten. Beide Ansätze werden als Kalibrierungsmöglichkeit angeboten, zunächst wird dafür das Messverfahren analysiert.

6.1.1 Zeitmessung durch Abfragen eines Hardware-Timers

Die Umsetzung des in Kapitel 4 beschriebenen Messprinzips ist in Abb. 6.1 als elektronische Schaltung dargestellt, zur Bestimmung der Messabweichung müssen die Elemente der Messeinrichtung betrachtet werden.

Der exemplarische Aufbau einer Zeitmesseinrichtung besteht aus Hilfsgeräten mit einem Oszillator, der einen Referenztakt mit der konstanten Frequenz f_R über einen Teiler auf einen Mengenzähler ausgibt, der synchron zum Takt des Oszillators dessen Impulse I zählt. Als weiteres Glied der Messkette dient das Messgerät, es wird durch eine Torschaltung realisiert, die über eine Auswahl einen Zählerwert auf einen der 1 bis n Speicher ausgibt. Als Torschaltung wird vereinfacht ein Demultiplex-Glied mit zwei Eingängen verwendet. Ein Eingang wird mit der Messgröße t_x belegt und der zweite mit dem Zählerwert n_x. Die Anbindung zum Monitorsystem erfolgt durch Auswahl des Speicherbereiches, der vom Messwerkzeug ausgelesen werden kann.

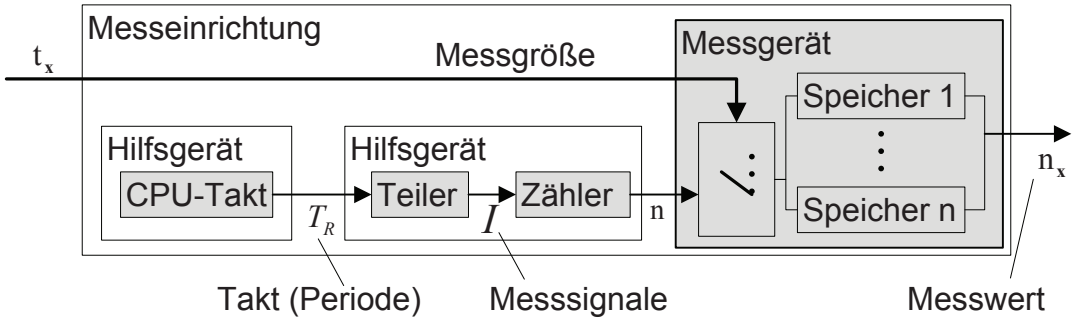


Abb. 6.1: Zeitmesseinrichtung

Annahmen: Der Takt mit konstanter Frequenz, der den Zähler steuert, entspricht dem Arbeitstakt der CPU (Timerzyklen = CPU-Zyklen bzw. Teiler 1/1). Damit wird eine Auflösung von einem CPU-Zyklus erreicht. Der verwendete Zähler besitzt eine ausreichend große Zählkapazität, so dass ein Überlauf während einer Messung nicht erfolgt. Diese Annahmen scheinen berechtigt, da für gängige Zeitbausteine das Teilerverhältnis konfigurierbar und die Zyk-

len über mehrere Jahre gezählt werden können (s. [10]). Die Rückwirkung des Monitorsystems beschränkt sich ausschließlich auf die Wahl des RAM-Bereiches, auf den das Messwerkzeug zugreifen kann (s. CAL-RAM Problematik im Kapitel 4).

Anmerkung: Die Zeiteffekte Drift und Diskretisierung[1], wie sie bei der Zeitnahme durch eine perfekte Uhr gegenüber einer realen Uhr festgestellt werden können, finden keine Berücksichtigung, d. h. eine Betrachtung der Oszillator-Qualität findet nicht statt. Zudem erfolgt nur eine indirekte Zählung des Arbeitstaktes der CPU. Die Zeitmessung bzw. Zeitintervallbestimmung (s. Abb. 6.2) folgt dabei aus der Wertedifferenz zweier Speicherstände (Zählerwerte) n_x und der Frequenz des Referenzoszillators f_R bzw. der Periodendauer T_R nach Gl. 4.1 und Gl. 4.2.

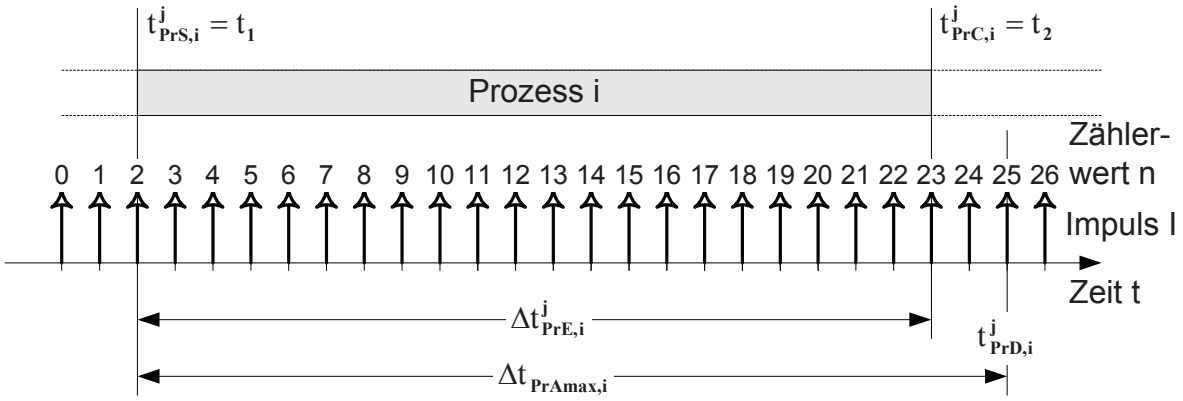


Abb. 6.2: Zeitintervallbestimmung

Beispiel: Die Berechnung der Prozesslaufzeit $\Delta t^j_{\mathrm{Pr}E,i}$ des Prozesses i für die j-te Ausführung nach Gl. 4.1 und Gl. 4.2 ergibt für das Beispiel aus Abb. 6.2 mit $t_1^j = n_1^j = 2$; $t_2^j = n_2^j = 23$: $\Delta t^j_{\mathrm{Pr}E\,i} = (n_2^j - n_1^j) * T_R = 21\,\text{Zyklen} * T_R$.

In den nachfolgenden Betrachtungen wird vorausgesetzt, dass die Hilfsgeräte der Messeinrichtung in der Hardware des Prozessors implementiert sind und die Implementierung des Messgerätes durch Software erfolgt. Hierbei ist die Torschaltung eine Analysemarke, die bei ihrer Ausführung den „aktuellen“ Zählerstand auf einen Speicher ausgibt. Hinzu kommt die Nummer der Analysemarke, die ebenfalls in einem Speicher abgelegt wird. In Tabelle 6.1 ist eine Beispielimplementierung von Analysemarke und Nummer (Messpunkt) als C-Code dargestellt.

6.1.2 Rückwirkung der Messeinrichtung auf das Messobjekt

Wie bereits in Kapitel 3 und 4 diskutiert wurde, besteht ein Messpunkt (MP) aus Befehlen, die zusätzlich zum Programm (dem Messobjekt - MO) abgearbeitet werden müssen. Damit unterliegt ein MP den gleichen Laufzeiteinflüssen wie das MO, hierzu zählen z. B. Cache-Fehlzugriffe. Die Laufzeit eines MP in einem Programm kann damit bei jedem Durchlauf unterschiedlich sein, insbesondere kann die Ausführung mit dem MO überlappen, wenn das Programm auf einer Hardwareumgebung mit Pipelinearchitektur, superskalaren Architektur usw. ausgeführt wird. Die Ausführung eines MP bringt ein weiteres Problem mit sich. Die Zeitnahme (z. B. durch Auslesen des Timers) erfolgt nicht unmittelbar vor dem Start und nach dem Ende der Ausführung des Messobjektes, da die einzelnen Befehle des MP abgearbeitet werden müssen. Dieser Effekt des Zeitversatzes ist zusätzlich zu dem bereits erwähnten Effekt, der Überlappung von MO und MP, zu berücksichtigen.

[1] Die Vernachlässigung der resultierenden Abweichungen erfolgt unter der Annahme, dass kein Teiler benutzt wird, da sonst eine Gleichverteilung des unbekannten Fehlers über die Periodendauer anzunehmen ist.

Dabei setzt sich der Zeitversatz aus zwei aufeinander folgenden Zeitspannen zusammen:

- der Zeitspanne von der ersten Zeitnahme bis zum Start der Ausführung des Messobjektes und
- der Zeitspanne, in der die nachfolgende Ausführung des Messobjektes ihren Abschluss findet, und der zweiten Zeitnahme.

Die Abb. 6.3 stellt die Zeitnahme zusammen mit der resultierenden Messgeräteabweichung dar, dabei kann die Zeitnahme wie folgt erklärt werden:

- B_k sei der Befehl der die Zeitnahme (z. B. Auslesen des Timers) zum Zeitpunkt t_0 auslöst.
- $B_{k+1}, \ldots, B_n$ seien die Befehle, die nach der Zeitnahme (1) (Bereitstellung) des Zeitwertes (t_0) verwendet werden. Bezogen auf das Messobjekt wird mit B_{n+1} zum Zeitpunkt t_1 die Ausführung des Messobjektes gestartet. Das bedeutet jedoch, dass anstelle des Ereignisses (t_1,B_{n+1}) zum Zeitpunkt t_1 in der Zeitmessung das Ereignis (t_0,B_k) verwendet wird.
- $B_1, \ldots, B_k$ seien die Befehle, die bis zur Zeitnahme (2) (Vorbereitung) des Zeitwertes (t_3) verwendet werden. Bezogen auf das Messobjekt wird mit B_{l-1} zum Zeitpunkt t_2 die Ausführung des Messobjektes abgeschlossen. Das bedeutet jedoch, dass anstelle des Ereignisses (t_2,B_{l-1}) zum Zeitpunkt t_2 in der Zeitmessung das Ereignis (t_3,B_k) verwendet wird.

Wie aus dem Ablauf der Zeitnahme hervorgeht, findet immer eine Laufzeitmessung zwischen zwei Messpunkten (zwei Einzelmesswerten) statt, diese werden im Folgenden als Messpunktpaar (MPP) bezeichnet. Zur Berücksichtigung des Zeitversatzes wird eine innere MPP-Laufzeit definiert:

Definition 6.1 MPP-Laufzeit: Δt_{MPP}**:** Die innere Laufzeit eines Messpunktpaares (MPP-Laufzeit) Δt_{MPP} im gerade ausgeführten Programmpfad ist die Laufzeit, die nach der Ausführung des ersten Befehls zur Zeitnahme beginnt und mit der Ausführung des nächsten Befehls zur Zeitnahme im Programmpfad endet, ohne die Zeit, die das Messobjekt zur Ausführung benötigt.

Die MPP-Laufzeit muss als Messgeräteabweichung betrachtet werden. Um die Abweichung zu bestimmen, werden die Einzelabweichungen aus Vorbereitungs- und Bereitstellungszeit betragsmäßig addiert. Die Einzelabweichungen setzen sich aus bekannter und unbekannter systematischer und zufälliger Messabweichung zusammen.

Beispiel: Zur Veranschaulichung soll der Befehlscode in Tabelle 6.1 dienen, der zu den Messpunkten 1000 und 1001 vom Compiler erzeugt wurde. Die durch das MPP verursachte Messabweichung kann, wie in Abb. 6.3 dargestellt, auf MP 1000 als Bereitstellungszeit und MP 1001 als Vorbereitungszeit berücksichtigt werden.

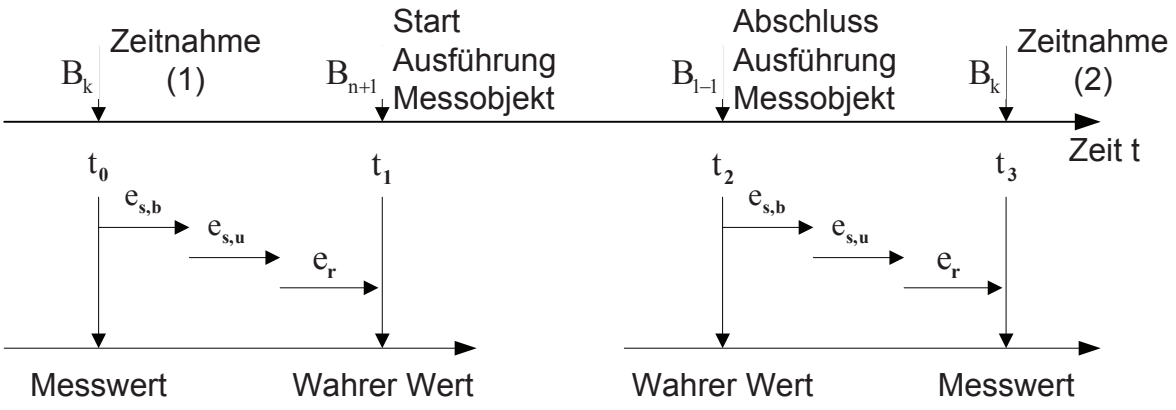


Abb. 6.3: Messabweichung durch Instrumentierung

Tabelle 6.1: Messpunkt C-Code und Assembler

Messpunkt C-Code	Befehl	Assembler GNU-TriCore
tmpVWMP_GEN[MP_INDEX++] = 1000;		movh.a %a2,HI:MP_INDEX
//Speichern der Messpunktnummer 1000		movh.a %a3,HI:tmpVWMP_GEN
tmpVWMP_GEN[MP_INDEX++] =		ld.w %d15, [%a2] LO:MP_INDEX
(*(volatile uint32*)0xF0000210);		sha %d0, %d15, 2
//Auslesen des Systemtimers mit Adresse 0xF0000210		lea %a3,[%a3] LO:tmpVWMP_GEN
		add %d15, %d15, 1
		addsc.a %a15, %a3, %d0, 0
		movh.a %a3,HI:tmpVWMP_GEN
		sha %d0, %d15, 2
		add %d15, %d15, 1
		st.w [%a2] LO:MP_INDEX, %d15
		mov %d15, 1000
		st.w [%a15] 0, %d15
		lea %a3,[%a3] LO:tmpVWMP_GEN
	B_k	ld.w %d15, 0xf0000210
	.	addsc.a %a15, %a3, %d0, 0
	B_n	st.w [%a15] 0, %d15
tmpVWMP_GEN[MP_INDEX++] = 1001;	B_1	movh.a %a3,HI:tmpVWMP_GEN
//Speichern der Messpunktnummer 1001	.	ld.w %d15, [%a2] LO:MP_INDEX
tmpVWMP_GEN[MP_INDEX++] =	.	sha %d0, %d15, 2
(*(volatile uint32*)0xF0000210);	.	add %d15, %d15, 1
//Auslesen des Systemtimers mit Adresse 0xF0000210	.	lea %a3,[%a3] LO:tmpVWMP_GEN
	.	addsc.a %a15, %a3, %d0, 0
	.	sha %d0, %d15, 2
	.	add %d15, %d15, 1
	.	st.w [%a2] LO:MP_INDEX, %d15
	.	mov %d15, 1001
	.	st.w [%a15] 0, %d15
	.	addsc.a %a15, %a3, %d0, 0
	B_k	ld.w %d15, 0xf0000210
		st.w [%a15] 0, %d15

6.2 Methoden zur Bestimmung der MPP-Laufzeit

Grundsätzlich können zwei Methoden zur Ermittlung der bekannten systematischen Abweichung, die durch ein MPP verursacht wird, unterschieden werden:

1. theoretische Betrachtung (s. Abschn. 6.2.1) und
2. Referenzlaufzeitmessung (s. Abschn. 6.2.2).

6.2.1 Theoretische Betrachtung

Die Analyse der bekannten Bedingungen der Messung, die sich unter anderem aus den Befehlen des MPP (Tabelle 6.2) ableiten lassen, kann zu einer theoretischen Betrachtung der möglichen Ursachen für Messgeräteabweichungen und zur Berechnung der bekannten systematischen Abweichung benutzt werden. Dafür wird die MPP-Laufzeit Δt_{MPP} durch Summierung der Ausführungszeiten der Befehle, die die Definition 6.1 erfüllen, bestimmt.

Annahme: Das zu analysierende MPP wird auf einer bestimmten Hardwareumgebung ausgeführt. Eine Tabelle mit allen Befehlsausführungszeiten (engl. Instruction Timing Table - ITT) für die Hardware enthält für die Befehle B_{k+1} bis B_n und B_1 bis B_k die Laufzeiten $t_{Befehl\,i}$, $i = 1,2,...,n$.

Die MPP-Laufzeit Δt_{MPP} wird dann wie folgt berechnet:

$$\Delta t_{MPP}^{ITT} = \underbrace{\sum_{k+1}^{n} t_{Befehl\,i}}_{Zeitnahme\,1} + \underbrace{\sum_{1}^{k} t_{Befehl\,i}}_{Zeitnahme\,2} \tag{6.5}$$

Bei heutigen komplexen Prozessoren sind Architekturmerkmale, wie parallele Ausführung, Pipeline und Cache, zu finden, die wesentlichen Einfluss auf die MPP-Laufzeit haben. Die Laufzeitbestimmung durch Summierung der Befehlsausführungszeit führt dann zu unbekannten systematischen und zufälligen Messabweichungen. Am Beispiel einer typischen Prozessorfamilie für eine Motorsteuerung soll dies im Folgenden erläutert werden.

Motorsteuerung

Der hier betrachtete Mikrocontroller ist der TriCore 1 von Infineon [10]. Der TriCore 1 ist ein Prozessor mit Superskalarität. Um dieses Ziel zu erreichen, besitzt der Prozessorkern drei getrennte Verarbeitungseinheiten. Die Pipelinearchitektur unterteilt sich in zwei Hauptpipelines und eine Nebenpipeline. Zu den Hauptpipelines gehört die Integer-Einheit (I-Pipeline). Diese Einheit bearbeitet alle arithmetischen Befehle sowie die bedingten Sprünge auf Datenregister. Sie umfasst die arithmetisch-logische Einheit (ALU), den Barrelshifter und die MAC-Einheit (Multiply and Accumulate). Die zweite Hauptpipeline ist die Load-Store-Einheit (L/S-Pipeline). Diese Einheit verarbeitet alle Load-Store- und Adressarithmetikoperationen, unter anderem auch Sprungbefehle und Kontextwechselbefehle. Weiterhin werden all jene Befehle von der Load-Store-Einheit verarbeitet, die sowohl auf Daten- als auch auf Adressregister zugreifen (z. B. die mov-Befehle und auch die konditionalen Befehle). Die dritte Pipeline ist die Loop-Einheit, sie wird als Nebenpipeline bezeichnet. Diese Einheit verarbeitet die Zero-Overhead-Loop-Befehle. Die Hauptpipelines sind vierstufig aufgebaut und bestehen aus der Fetch-, Decode-, Execute- und Write-Back-Phase. In der Fetch-Phase wird ein Befehl abgeholt. Bei der Decode-Phase wird der zuvor abgeholte Befehl decodiert und die erforderlichen Operanden abgeholt. In der Execute-Phase wird der spezifizierte Arbeitsgang mithilfe der abgeholten Operanden ausgeführt. In der Write-Back-Phase wird das Ergebnis in die spezifizierten Speicherbereiche geschrieben.

In der Instruction Timing Table des TriCores 1-V1.3 (TC1796) [10] wird vom allgemeinen Fall ausgegangen. Dabei wird angenommen, dass die Befehle unabhängig ausgegeben werden, d. h., es werden keine Datenabhängigkeiten oder Hasards zwischen den Befehlen betrachtet. Der TriCore ist so aufgebaut, dass bis auf wenige Ausnahmen die Befehle in einem Takt-Zyklus ausgeführt werden können (Takt-Zyklus entspricht hier dem CPU-Zyklus). Die Ausnahmen bilden Multiplikation und Verzweigungsbefehle, die mehr als einen Zyklus benötigen. Zusätzlich existiert eine kleine Zahl von selten verwendeten Befehlen, die ebenfalls mehr als einen Zyklus benötigen.

Beispiel: Die Berechnung der MPP-Laufzeit Δt_{MPP}^{ITT} nach Gl. 6.5 ergibt für das Beispiel aus Tabelle 6.1 bei einer Befehlsauführungszeit $t_{Befehl\,i} = 1$ Zyklus (lt. ITT TriCore) für die verwendeten Befehle $i = 1,2,...,n$ eine MPP-Laufzeit von $\Delta t_{MPP}^{ITT} = 15$ Zyklen.

Am Beispiel des TriCores werden im Folgenden die resultierenden Messunsicherheiten diskutiert, die zu unbekannten systematischen Messabweichungen führen.

Nebenläufigkeit: Befehle können teilweise parallel ausgeführt werden. Die ITT berücksichtigt jedoch nur Einzelbefehlausführungen.

Beispiel: Von der gemeinsam genutzten Fetch-Einheit können im Idealfall innerhalb eines Taktzyklus beide Hauptpipelines mit einem Befehl befüllt werden. Voraussetzung dafür ist jedoch, dass im Programmcode zuerst ein Integer-Pipeline- und dann ein Load/Store-Pipeline-

Befehl direkt hintereinander stehen (s. Tabelle 6.2). Für die Reihenfolge erst Load/Store-Pipeline- und dann ein Integer-Pipeline-Befehl wird keine parallele Ausgabe erreicht (s. Tabelle 6.3). Solch eine Kombinationsabhängigkeit nennt man paired instructions.

Tabelle 6.2: Befehlsequenz ADD d15, d15, 0x1; LEA a3, [a3] 0x1894; (s. Tabelle 6.1)

Pipeline	Einheit	1.	2.	3.	4.	5. Takt
I-Pipeline	IF	ADD				
	DC		ADD			
	EX			ADD		
	WB				ADD	
L/S-Pipeline	IF	LEA				
	DC		LEA			
	EX			LEA		
	WB				LEA	

Tabelle 6.3: Befehlsequenz LD.W d15, [a2] 0x1890; SHA d0, d15, 0x2; (s. Tabelle 6.1)

Pipeline	Einheit	1.	2.	3.	4.	5. Takt
I-Pipeline	IF		SHA			
	DC			SHA		
	EX				SHA	
	WB					SHA
L/S-Pipeline	IF	LD				
	DC		LD			
	EX			LD		
	WB				LD	

Ergebnis: Die berechnete MPP-Laufzeit kann höher sein als die reale Laufzeit, da für Befehle, die parallel ausgeführt werden, nicht unbedingt zusätzliche Zeit benötigt wird. Wird die MPP-Laufzeit zur Korrektion von der gemessen Laufzeit abgezogen, bedeutet das, dass die Messobjektlaufzeit unterschätzt wird.

Pipelining: Auf Pipelining basierende Systeme unterliegen Struktur-, Kontroll- und Datenabhängigkeits-Konflikten (Hasards). Befehle können Hasards verursachen, deren Auflösung Zeit in Anspruch nimmt, diese Hasards werden in der ITT nicht betrachtet.

Beispiel: Struktur-Hasards sind das Ergebnis von Ressourcekonflikten, bei dem auf eine Ressource durch mehr als einen Befehl gleichzeitig zugegriffen werden soll (s. Tabelle 6.4).

Tabelle 6.4: Befehlsequenz ADD d0, d1, d2; LD d0, [a0]0; (s. [12])

Pipeline	Einheit	1.	2.	3.	4.	5. Takt
I-Pipeline	IF	ADD				
	DC		ADD			
	EX			ADD		
	WB				ADD	
L/S-Pipeline	IF	LD				
	DC		LD			
	EX			LD		
	WB				LD	

Sowohl der ADD- als auch der LD-Befehl, die parallel ausgeführt werden, nehmen im 4. Takt dasselbe Zielregister in Anspruch (Ressourcekonflikt). Der *Write After Write* (WAW) Hasard wird in der Decodier-Einheit entdeckt und wird durch einen Wartezyklus (Stall-Zyklus) in der Pipeline entfernt (s. Tabelle 6.5).

T**abelle 6.5:** Befehlsequenz ADD d0, d1, d2; LD d0, [a0]0; (s. [12])

Pipeline	Einheit	1.	2.	3.	4.	5. Takt
I-Pipeline	IF	ADD				
	DC		ADD	-		
	EX			ADD	-	
	WB				ADD	-
L/S-Pipeline	IF	LD				
	DC		LD	LD		
	EX			-	LD	
	WB				-	LD

Anmerkung: '-' zeigt einen Stall-Zyklus (nop) an

Vergleichbar mit der gezeigten Abhängigkeit zweier parallel ausgeführter Befehle muss diese Abhängigkeit auch innerhalb einer Pipeline betracht werden. Die Auswirkungen von Abhängigkeiten von verschiedenen Befehlen auf die MPP-Laufzeit müssen berücksichtigt werden. Daten-Hasards spielen nur innerhalb eines MP eine Rolle, verursacht durch Befehle, die vom Ergebnis eines vorangegangen Befehls in einer Weise abhängen, die sich durch die überlappende Abarbeitung herausstellt. Eine Datenabhängigkeit zwischen MP und MO wird ausgeschlossen, da der MP unabhängig vom MO sein muss. Der TriCore beschränkt den Einfluss von Datenabhängigkeiten innerhalb der Pipeline in dem „forwarding“ Pfade verwendet werden. Diese „forwarding“ Pfade erlauben das Ergebnis eines Befehls zu den Eingängen eines Folgebefehls zu übergeben, ohne auf das abschließende Schreiben zu warten.

Ergebnis: Die berechnete MPP-Laufzeit kann durch Pipelining kleiner sein als die reale Laufzeit. Das bedeutet, dass die Messobjektlaufzeit überschätzt wird. Steuer-Hasards müssen im MP nicht auftreten, da auf Verzweigungen und andere Befehle, die den PC ändern, im MP verzichtet werden kann (s. Tabelle 6.1).

Operandenzugriff: Befehle können auf Daten zugreifen, deren Wertebereitstellung über mehrere Zyklen andauern kann. In der ITT wird vom besten Fall ausgegangen, d. h. von Daten, die im Cache (oder innerem SRAM) verfügbar sind und auf die deshalb in einem Takt-Zyklus zugegriffen werden kann.

Beispiel: Mit dem Befehl (B_k) „ld.w %d15, 0xf0000210“ wird auf den Systemtimer zugegriffen. Laut ITT benötigt ld.w nur einen Zyklus, in der Berechnung fehlen jedoch die Zyklen für den Peripheriezugriff auf den Systemtimer.

Ergebnis: Die berechnete MPP-Laufzeit kann kleiner sein als die reale Laufzeit. Das bedeutet, dass die Messobjektlaufzeit überschätzt wird.

Speicherhierarchien: Wie bereits beschrieben, beruhen alle Angaben in der ITT auf Daten, die im Cache (oder innerem SRAM - iRAM) verfügbar sind und auf die deshalb innerhalb eines Takt-Zyklus zugegriffen werden kann. Wenn die Daten (oder Befehle) nicht im Cache sind, dann werden in der Pipeline Wartezyklen eingefügt, bis die Daten verfügbar sind. Abhängig von der Implementierung kann das mehrere Zyklen zur Ausführung jedes Befehls hinzufügen. Das heißt, in der berechneten MPP-Laufzeit werden keine Cache-Fehlzugriffe oder andere Wartezyklen aus den Speicherhierarchien berücksichtigt, real sind jedoch Fehlzugriffe zu erwarten, was zu höheren Laufzeiten führt.

Fazit: Parallelität, datenabhängige Befehlsausführungszeit, Pipelining und Speicherhierarchien sind aufwendig zu bestimmen und setzen genaue Kenntnisse der Architektur voraus. Eine Bestimmung der systematischen Messabweichung ist auf diesem Wege äußerst schwierig und mit sehr hohem Aufwand verbunden.

6.2.2 Referenzlaufzeitmessung

Eine weitere Möglichkeit der Ermittlung der bekannten systematischen Abweichung bietet die Referenzlaufzeitmessung. Für diese Referenzlaufzeitmessungen sind dann Messeinrichtungen zu verwenden, die eine wesentlich kleinere Messabweichung als die zu analysierende Messeinrichtung besitzen.

Kalibrierung durch Referenzmessobjekt: Eine Kalibrierung kann durch das Anlegen einer genau bekannten Laufzeit Δt_r an einen Zeitmesser und die Berechnung der Differenz des Anzeigewertes des Zeitmessgerätes Δt_a vom wahren bzw. richtigen Wert Δt_r mit:

$$\Delta t = \Delta t_a - \Delta t_r \tag{6.6}$$

erfolgen. Der Bezugswert Δt_r kann dazu mit einem Referenzlaufzeitmessgerät bestimmt worden sein.

Beispiel: Ein Referenzlaufzeitmessgerät ist der Logic-State-Analyser (LSA) wie er im Kapitel 3 vorgestellt wurde. Für die zyklusgenaue Protokollierung der ablaufenden Adressdaten mittels LSA kann angenommen werden, dass hier eine vernachlässigbare Abweichung vom wahren Wert der Messgröße vorliegt. Die Tabelle 6.6 zeigt die Anzeigewerte einer Laufzeitmessung durch Messpunktinstrumentierung und durch LSA mit eingefügten Assembler-Befehlen. Beim ersten Versuch wurden 80 *nop* Befehle zwischen den Messpunkten eingefügt, beim zweiten 80 add Befehle. Die genau bekannte Laufzeit Δt_r ergibt sich aus dem Anzeigewert des LSA.

Tabelle 6.6: Laufzeitmessung

Bestimmungsmethode	LSA	Messung	LSA	Messung
Messobjekt	80 nop	80 nop	80 add	80 add
Befehlsspeicher	SPRAM	SPRAM	SPRAM	SPRAM
Datenspeicher	iRAM	iRAM	iRAM	iRAM
Anzeige Δt_a (in Zyklen)	80	100	80	99

Wird die Zeit Δt aus Gl. 6.6 als MPP-Laufzeit Δt_{MPP} interpretiert, so zeigt das Beispiel eine resultierende Messunsicherheit abhängig von den Befehlen, die zwischen dem MPP ausgeführt werden.

$\Delta t_{a,80nop}^{Messung} = 100;\ \Delta t_{a,80nop}^{LSA} = 80;\ \Delta t_{nop} = \Delta t_a - \Delta t_r = 20\,\text{Zyklen}$

$\Delta t_{a,80add}^{Messung} = 99;\ \ \Delta t_{a,80add}^{LSA} = 80;\ \Delta t_{add} = \Delta t_a - \Delta t_r = 19\,\text{Zyklen}$

Dieser Effekt ist auf die beschriebene Nebenläufigkeit beim Übergang vom MO auf den MP zurückzuführen, wenn paired instructions vorhanden sind.

Kalibrierung durch Referenzmesspunktpaar: Eine weitere Möglichkeit der Referenzlaufzeitmessung bietet die explizite Untersuchung des MPP. Um die MPP-Laufzeit Δt_{MPP} und die Anordnung der Befehle in der Pipeline zu bestimmen, wird die Laufzeit aller MPP-Befehle mit einem LSA aufgenommen. Aus dem erzeugten Trace können Rückschlüsse auf die Programmausführung gezogen werden. Hierbei interessiert besonders, welche der Pipelines von den Befehlen belegt werden (z. B. Load/Store- oder Integer-Pipeline) und wie lange es dauert, bis der nächste Befehl in der Pipeline angeordnet werden kann. Damit lassen sich Nebenläufigkeiten, Operandenzugriffe und Auswirkungen von Speicherhierarchien auf die MPP-Laufzeit beurteilen.

Beispiel: Das zu untersuchende MPP wird unter unterschiedlichen Konfigurationen auf dem Zielsystem ausgeführt, es wird jeweils ein Trace aufgezeichnet. Dabei werden die Auswirkungen von Cache, Pipeline und das Auslagern der Messvariablen in den externen RAM (e-

RAM) des Zielsystems auf die MPP-Laufzeit untersucht werden. Mit dem Vergleich von i-RAM und eRAM wird die Rückwirkung durch das Monitorsystem (s. Kapitel 4) zwischen RAM- und CAL-RAM-Bereich untersucht.

Tabelle 6.7: Trace zweier Messpunkte (Befehls-/Datenspeicher - SPRAM/iRAM)

Index	Trace	Pipeline	Befehl
19	LD.W d15, 0xf0000210	L/S	B_k
20	ADDSC.A a15, a3, d0, 0	L/S	.
21	ST.W [a15] 0, d15	L/S	B_n
22	MOVH.A a3, 0xd000	L/S	B_1
23	LD.W d15, [a2] 0x1890	L/S	.
24	SHA d0, d15, 0x2	I	.
25	ADD d15, d15, 0x1 | LEA a3, [a3] 0x1894	I | L/S	.
26	ADDSC.A a15, a3, d0, 0	L/S	.
27	SHA d0, d15, 0x2	I	.
28	ADD d15, d15, 0x1 | ST.W [a2] 0x1890, d15	I | L/S	.
29	MOV d15, 0x3e9 | ST.W [a15] 0, d15	I | L/S	.
30	ADDSC.A a15, a3, d0, 0	L/S	.
39	LD.W d15, 0xf0000210	L/S	B_k

Der Trace (s. Tabelle 6.7) wurde aufgenommen ab dem Auslesebefehl des Systemtimers vom ersten der zwei Messpunkte. An dieser Stelle startet auch die Zeitmessung des Messverfahrens. Der Index ist vom CPU-Takt abgeleitet, sodass die Differenz der Indizes direkt zu den benötigten Taktzyklen führt. Die erste Zeitnahme beginnt bei Index 19. An dieser Stelle ist der Systemtimer erstmalig ausgelesen, bei Index 39 das zweite Mal. Die Differenz 39-19 ergibt genau die gemessenen 20 Zyklen.

Tabelle 6.8: Laufzeitbedarf der Instrumentierung

Bestimmungsmethode	LSA	LSA	LSA	LSA	ITT
Befehlsspeicher	SPRAM	SPRAM	iROM	iROM	Cache
Datenspeicher	iRAM	eRAM	iRAM	eRAM	iRAM
Anzeige Δt_{MPP} (in Zyklen)	20	30	23	35	15

Aus den Messergebnissen der Tabelle 6.8 geht hervor, dass die benötigte MPP-Laufzeit zwischen 20 und 35 Prozessorzyklen beträgt, je nachdem, ob sich die Befehle für den Messpunkt im SPRAM (vergleichbar mit 100% Cache-Hit) oder im iROM (vergleichbar mit 100 % Cache-Miss) befinden und ob die Messwerte in den externen oder internen RAM abgelegt werden müssen. Zum Vergleich ist noch einmal die berechnete MPP-Laufzeit (nach ITT) angegeben.

6.3 Bewertung der Messgeräteabweichung

Die unterschiedlichen MPP-Laufzeiten zeigen, dass das Messverfahren auf dem Steuergerät nur sinnvolle Messergebnisse liefern kann, wenn die Messgeräteabweichung bestimmt ist. Die bekannte systematische Messabweichung kann, wie unter Abschn. 6.2.1 oder 6.2.2 dargestellt, mit den vorhandenen Mitteln, aber einem verfahrensabhängigen nicht unerheblichen Aufwand ermittelt werden. Das bedeutet einen hohen Analyseaufwand sowie die notwendige Verfügbarkeit von hochauflösenden LSA mit entsprechender Kanalbreite. Dies sind Gründe,

die gegen eine Verwendung des LSA als Referenzlaufzeitmessgerät sprechen. Die ermittelten MPP-Laufzeiten gelten streng genommen jeweils nur für den betrachteten Kontext (Speicher, Compiler usw.). Im Versuch konnte unter Zuhilfenahme des Trace einer Messung zweier aufeinanderfolgender Messpunkte gezeigt werden, wie die zur Ausführung der Instrumentierung benötigte Anzahl an CPU-Zyklen zustande kommt. Gleichzeitig ist aus der Anordnung der Befehle in der LS- und I-Pipeline erschließbar, dass ein Messobjekt, das mit einem I-Befehl endet (d. h., I-Befehl vor dem Messpunktbefehl B_1), bei einer Messung aufgrund von Nebenläufigkeit nicht gemessen wird.

6.3.1 Bekannte systematische Messgeräteabweichung

Die Erkenntnisse der vorherigen Abschnitte lassen folgende Schlussfolgerungen zu:

Als bekannte systematische Messabweichung kann die untere Grenze der gemessenen MPP-Laufzeit angegeben werden. In einer Laufzeitmessung ist diese Rückwirkung auf das Messobjekt immer enthalten.

$$e_{Ms,b} = \min\left(\Delta t_{MPP}\right) \tag{6.7}$$

Beispiel: Bei Berücksichtigung der im Abschn. 6.2.1 beschriebenen TriCore Merkmale (Befehlsspeicher SPRAM - Zugriff innerhalb eines Zyklus, Datenspeicher iRAM - Zugriff innerhalb eines Zyklus und Peripheriezugriff auf den Systemtimer innerhalb von 9 Zyklen) ergibt: $e_{Ms,b} = \min\left(\Delta t_{MPP}\right) = 20$ Zyklen.

6.3.2 Unbekannte systematische Messgeräteabweichung

Verfahrensbedingt ergeben sich durch die Instrumentierung unbekannte systematische Messabweichungen im Zusammenhang mit Cache, Pipeline, Nebenläufigkeit und Compileroptimierung.

Beispiel: Die benötigte MPP-Laufzeit (Tabelle 6.8) hängt davon ab, ob die Befehle für den Messpunkt im Cache sind und ob die Messwerte in den externen oder internen RAM abgelegt werden müssen. Wird die bekannte systematische Messgeräteabweichung vom Anzeigewert Δt_{MPP} abgezogen, ergibt sich eine unbekannte Messabweichung im Bereich zwischen 0 und 15 Zyklen unter der Annahme, dass zufällige Messabweichungen ausgeschlossen werden können.

Unbekannte systematische Messabweichungen $e_{Ms,u}$ treten dabei in analoger Art und Weise in Erscheinung, wie die bereits betrachteten systematischen Messabweichungen. Das bedeutet, dass bei der Aufnahme von Messwerten unter Wiederholbedingungen stets eine identische unbekannte systematische Messabweichung anzunehmen ist. Eine Bestimmung durch theoretische Betrachtung oder Referenzlaufzeitmessung und damit eine Korrektur der Messwerte ist nicht möglich. Vielmehr ist ein korrigiertes Messergebnis bezüglich seiner systematischen Abweichung mit einer Unsicherheit behaftet. Die Rückwirkungen durch unbekannte systematische Messabweichungen lassen sich nur selten völlig verhindern und müssen daher durch geeignete Maßnahmen auf einen vertretbaren und beschreibbaren Anteil reduziert werden.

Maßnahmen zur Reduktion der unbekannten systematischen Messgeräteabweichung

Bei einem idealen Messpunkt ist die Messabweichung vollständig beschreibbar, d. h., konstanter Zeitversatz bzw. konstante Befehlsausführungszeit, bekannte Pipelinebelegung und bekannte Nebenläufigkeit einschließlich der Rückwirkungen auf das Messobjekt durch Überlappung.

Berücksichtigung von Compileroptimierungen, Pipelinebelegung, Nebenläufigkeit und Cache des MPP: Eine Möglichkeit das Verhalten hinsichtlich der unbekannten systematischen Messabweichungen zu verbessern, ist die einheitliche Implementierung der Messpunkte, die entweder im Assembler (aufwendig und architekturabhängig) oder als vorkompiliertes Objekt eingebunden werden. In diesem Fall kann die durch Compileroptimierung entstehende Unsicherheit (s. unterschiedliche Befehle der Messpunkte 1 und 2 in Tabelle 6.2) vermieden werden. Die Befehlsfolge B_1 bis B_n kann an einem Messpunkt bestimmt werden und ist dann im gesamten Programm für jeden Messpunkt gültig (s. Abb. 6.4). Das heißt, aus der Betrachtung eines Messpunktes kann auf die Laufzeit eines MPP geschlossen werden, dies schließt Pipelinebelegung und Nebenläufigkeit mit ein.

Konstruktiver Ansatz zur Problemlösung: Eine Lokatierung der Messpunkte in einem CPU-nahen Speicher, der ohne Cache auskommt, sorgt für Unabhängigkeit von Cacheeinflüssen und damit für minimale Befehlsausführungszeiten. Hierfür bietet sich der Speicher an, der für die Angaben in der ITT benutzt wurde (z. B. SPRAM - TriCore). Die Einbindung des Messpunktes in das Programm kann hier als Serviceroutine (vorkompiliertes Objekt) erfolgen. Unter diesen Vorgaben lässt sich ein Messpunkt entwickeln, dessen unbekannte systematische Messabweichung reduziert wird.

Berücksichtigung des Monitorsystems: Die Messunsicherheit, die durch die Ablage der Messwerte im eRAM oder iRAM verursacht wird, ist abhängig vom verwendeten Monitorsystem. Daher kann für die bekannte systematische Messgeräteabweichung eine Korrekturtabelle erstellt werden, mit der sich abhängig vom Monitorsystem die Messwerte korrigieren lassen. Unter Einhaltung der Maßnahmen zur Reduktion der unbekannten systematischen Messabweichung kann folgende Korrekturtabelle angegeben werden (s. Tabelle 6.9):

Tabelle 6.9: Korrekturtabelle Messgeräteabweichung

	Monitorsystem mit iRAM	Monitorsystem mit eRAM
$e_{Ms,b}$ (in Zyklen)	46	51
$e_{Ms,u}$ (in Zyklen)	1	1

Berücksichtigung von Nebenläufigkeit zwischen MO und MP: Bei paired instructions kann ein Zyklus gespart bzw. nicht mit gemessen werden, daher ergibt sich eine unbekannte systematische Messabweichung $e_{Ms,u}$ von einem Zyklus. Das heißt, abhängig vom Programmpfad des Messobjektes können paired instructions vorhanden sein oder nicht.

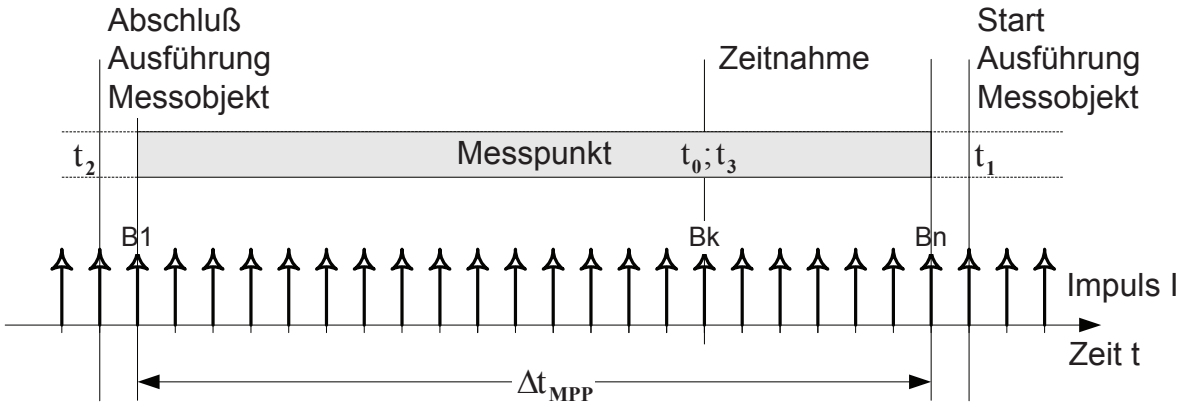


Abb. 6.4: Kontextunabhängiger Messpunkt

6.3.3 Berichtigung des Laufzeitmessergebisses durch Korrektion

Mit der ermittelten bekannten systematischen Messabweichung ist eine Berichtigung des unberichtigten Messergebnisses durch die Korrektion K möglich, wobei gilt:

$$e_{M\,s,b} = -K \tag{6.8}$$

Die Berichtigung ist über die Beziehung:

$$\Delta t_{Korr} = \Delta t + K \tag{6.9}$$

möglich. Hierbei stellen Δt_{Korr} das berichtigte Messergebnis und Δt das unberichtigte Messergebnis dar. Für das Beispiel aus Abb. 5.7 (s. Kapitel 5) ergibt sich das berichtigte Messergebnis in Abb. 6.5. Für das berichtigte Messergebnis ergibt sich aus der absoluten Messgeräteabweichung nach Tabelle 6.9 eine relative Messabweichung durch das Laufzeitmessgerät nach Gl. 6.4 von:

$$e_{Mrel}^{10\,\mu s} = \frac{e_M}{\Delta t_{\mathrm{Pr}\,A\,\max}^{10\,\mu s}} = \frac{1\,\mathrm{Zyklus}}{750\,\mathrm{Zyklen}} = 0{,}001$$

$$e_{Mrel}^{100\,\mu s} = \frac{e_M}{\Delta t_{\mathrm{Pr}\,A\,\max}^{100\,\mu s}} = \frac{1\,\mathrm{Zyklus}}{7500\,\mathrm{Zyklen}} = 0{,}0001$$

$$e_{Mrel}^{200\,\mu s} = \frac{e_M}{\Delta t_{\mathrm{Pr}\,A\,\max}^{200\,\mu s}} = \frac{1\,\mathrm{Zyklus}}{15000\,\mathrm{Zyklen}} = 0{,}00006$$

Das korrigierte Messergebnis beschreibt noch nicht die bestmögliche Angabe eines Messergebnisses. Dafür ist noch eine Ergänzung um die Messunsicherheit erforderlich, womit dann ein vollständiges Messergebnis erreicht wird. Dies wird im nächsten Abschnitt beschrieben.

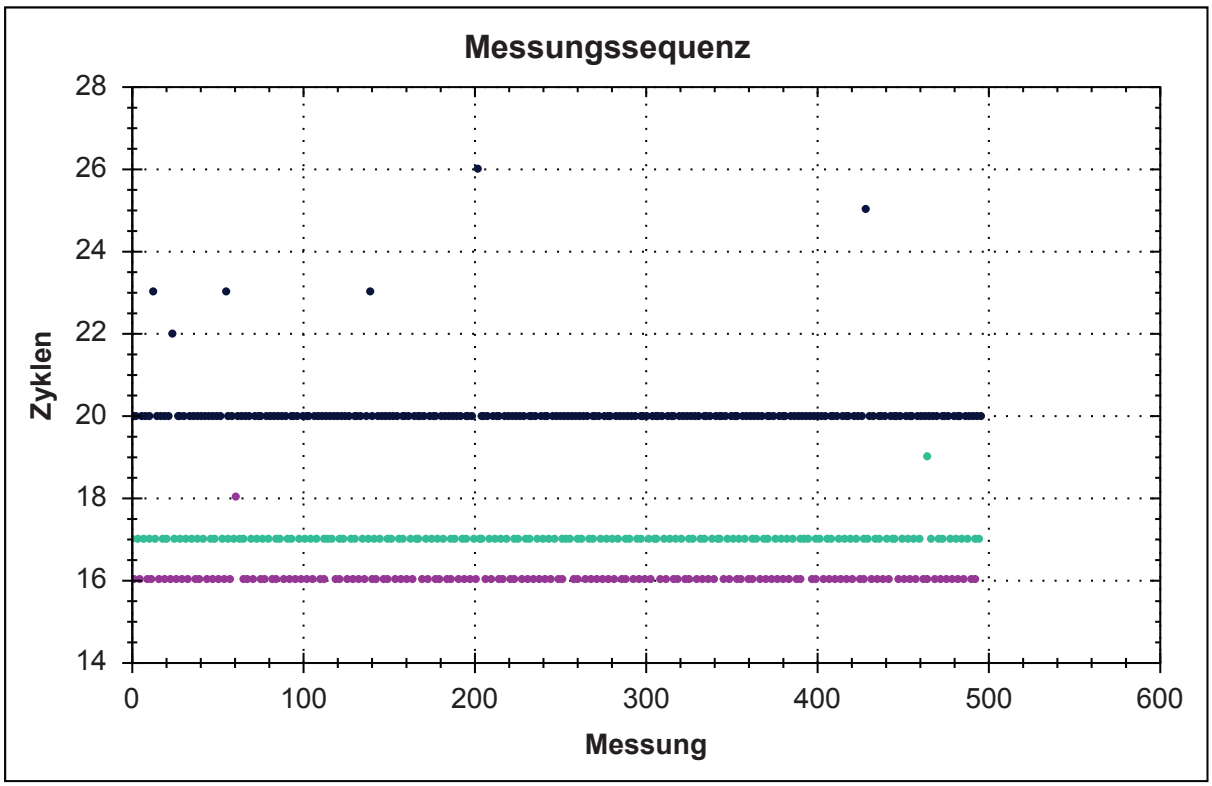


Abb. 6.5: Korrigierte Messergebnisse einer wiederholt durchgeführten Laufzeitmessung

6.3.4 Betrachtung zufälliger Messgeräteabweichung

Die durch zufällige Einflüsse verursachten Messabweichungen können nur durch Messungen unter weitestgehend identischen Bedingungen, den schon erläuterten Wiederholbedingungen, erkannt werden. Die im Ergebnis der wiederholten Messungen entstandene Messreihe wird zur Bewertung zufälliger Abweichungen üblicherweise mit statistischen Methoden analysiert. Eine geeignete mathematische Auswertung der erhaltenen Messreihe erlaubt dann Aussagen zu dem Bereich, in dem der wahre Wert bzw. der Erwartungswert der Messgröße mit einer angenommenen Wahrscheinlichkeit, also der vorgegebenen statistischen Sicherheit, liegt.

Beispiel: Die wiederholte Ausführung von Messungen unter identischen Bedingungen führt zu einer Messreihe, wie in Abbildung 6.5 dargestellt. Dabei werden beherrschbare Faktoren, wie der ausgeführte Programmpfad und Speicherbereich des Programms konstant gehalten. In Abb. 6.5 werden die Programmpfade farblich differenziert dargestellt, das Programm wird aus dem SPRAM (entspricht 100 % Cache-Hit) ausgeführt. Die hier beschriebene Vorgehensweise basiert auf einer statistisch ausreichend großen Messreihe.

Zur Auswertung der Messreihe werden die ermittelten Messergebnisse in ein Histogramm eingetragen (s. Abb. 6.6). Als Richtwert für die Klassenbreite wird die Auflösung des Hardware-Timers benutzt, für die Beispielmessreihe bedeute das $\Delta t = 1\ Zyklus$. Das Klassenintervall hat dabei folgende Einteilung $(0 \le \Delta t < 1; 1 \le \Delta t < 2; \ldots; n-1 \le \Delta t < n)$, wobei nur der obere Klassenwert im Histogramm dargestellt wird. Im Histogramm werden in Klammern die Programmpfadnummer (Pfad-ID) und deren Häufigkeit angegeben. Das Ergebnis weiterer Überlegungen ist dann üblicherweise die Verteilungsdichtefunktion und hieraus die Berechnung der Wahrscheinlichkeit, mit der ein Messergebnis $\Delta t_{\mathrm{Pr}\,E\,i}^{Pfad-ID}$ für einen Programmpfad in das durch $\Delta t_1 \le \Delta t < \Delta t_2$ begrenze Intervall fällt.

Der hier vorgeschlagene Ansatz betrachtet jedoch die Ursachen der Messabweichung und soll dazu genutzt werden, einen zweckmäßigeren Ansatz zur Behandlung der zufälligen Abweichungen zu ermöglichen.

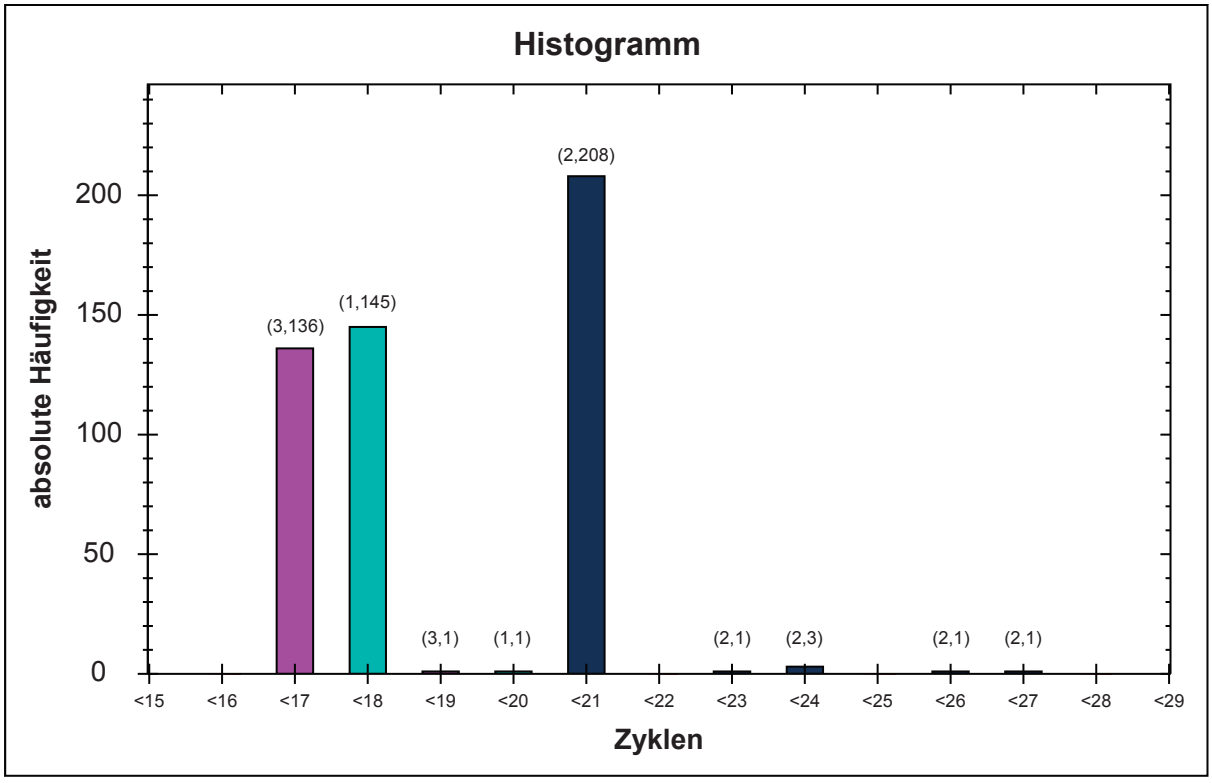


Abb. 6.6: Diskrete Messwertverteilung der Messreihe entsprechend Abb. 6.5

6.3.5 Behandlung zufälliger Messgeräteabweichung

Durch Betrachtung der zufälligen Messabweichung mit den Mitteln der Referenzlaufzeitmessung nach Abschn. 6.2.2 können die Ursachen der Laufzeitschwankungen eingegrenzt werden. Im Beispielprogramm (s. Abb. 6.6) ist der Grund für die unterschiedliche Anzahl an CPU-Zyklen eines unter Wiederholbedingungen betrachteten Programmpfades beim Messpunkt zu suchen. Nach der Betrachtung des Trace können die zusätzlichen Zyklen auf den Ladevorgang des Timerwertes zurückgeführt werden. Für das Auslesen des Timers wird bedingt durch das Chiplayout über die *LFI-Bridge* zwischen *Data Local Memory Bus (DLMB)* und *System Peripherals Bus (SPB)* auf den am SPB angeschlossenen Timer zugegriffen. Für die Zeit des Zugriffs steht die Pipeline in der Decode-Phase (2. Stufe) still (*Stall*). Eine Schwankung (Jittern) der Zeitdauer des Zugriffs auf den Systemtimer und allgemein den SPB ist laut [10] auf weitere Busmaster des SPB zurückzuführen.

Ansatz zur Behandlung zufälliger Messgeräteabweichung: Für zufällige Messabweichung, die auf das Messgerät zurückzuführen sind, kann die korrigierte Messobjektlaufzeit je Programmpfad wie folgt bestimmt werden.

$$\Delta t_{\mathrm{Pr}\,BCET}^{Pfad-ID} = \min\left(\Delta t_{\mathrm{Pr}\,E}^{Pfad-ID\ j}\right) \tag{6.10}$$

Der Ansatz vernachlässigt zufällige Abweichungen. Im gezeigten Beispiel ist dies gerechtfertigt, da die Abweichungen auf das Messgerät zurückzuführen sind. Der längere Zugriff auf den Timer erhöht die Laufzeit des Messpunktes, nicht die des Messobjektes.

Anmerkung: In der Praxis ist jedoch eine Unterscheidung zwischen zufälligen Messabweichungen e_r und der darin enthaltenen zufälligen Messgeräteabweichungen e_{Mr} nicht möglich. Hier bietet es sich an, zufällige Messgeräteabweichungen wie zufällige Messabweichungen zu behandeln.

Die Frage ist nun, ob der Ansatz zur Behandlung zufälliger Messgeräteabweichung auch dazu verwendet werden kann, um zufällige Messabweichungen, die nicht auf das Messgerät zurückzuführen sind, wie z. B. Cache- und Multitasking-Effekte, zu beurteilen. Dafür werden im Folgenden mehrere Messreihen betrachtet, die die Software-Funktion aus Abschnitt 5.2 unter unterschiedlichen Bedingungen untersucht.

6.4 Experimente zur Bewertung von Laufzeitmessergebnissen

Das Ziel der dargestellten Experimente und Analysen ist es, eine mathematische Analyse von Messreihen zu ermöglichen, die von zufälligen Einflüssen, wie Interruptbehandlung, Cache- und Multitasking-Effekten, überlagert sind. Dafür sollen die durch diese Einflüsse verursachten Messabweichungen betrachtet und daraus die Messergebnisse der Messreihe hinreichend genau charakterisiert werden.

Um den Einfluss eines Echtzeitbetriebssystems auf die Ausführungszeit einer Software-Funktion bestimmen zu können, müssen die Mechanismen bekannt sein, die das Betriebssystem verwendet. Zum einen muss die Interruptbehandlung [3, 5] bekannt sein, d. h., wann wird das Messsegment von der Interrupt-Service-Routine unterbrochen und wann wird das Segment wieder ausgeführt. Es muss der Einfluss durch die Verdrängung des Tasks aus dem Cache durch einen anderen Task beachtet werden. Wesentlich ist hierbei die maximale Strafzeit, die aus der Veränderung des Cache resultiert [31].

Für die Behandlung von Unterbrechungen bedeutet das erstens, die Unterbrechungen während des Programmablaufes mit zu messen und anschließend von der Segmentlaufzeit abzuziehen. Das setzt die Möglichkeit voraus, dass die Unterbrechungslaufzeit gesondert erfasst

werden kann oder dass dessen Laufzeit bekannt ist. Zweitens besteht gegebenenfalls die Möglichkeit, Unterbrechungen während der Programmlaufzeit zu deaktivieren.

In den Messungen B1 bis B3 werden die Testfälle aus Messung A2 (s. Kapitel 5) wiederholt. Wie in der Messung A2 wird die gemessene Prozesslaufzeit durch Laufzeiten anderer zeitsynchroner Tasks (s. Kapitel 2) auch in den Messungen B1 und B3 überlagert. In den Messungen B1 und B3 kommen noch drehzahlsynchrone Tasks hinzu (Drehzahl: n = 4500 min^{-1}), die zusätzlich zu Unterbrechungen führen. Um zu bewerten, welchen Einfluss die unterbrechenden Tasks haben, wird die Messung B2 ohne Unterbrechungen durchgeführt. Mit der Messung B3 sollen Cache-Effekte beurteilt werden, hier werden nicht die Interrupts deaktiviert, sondern das Programm ohne Cache ausgeführt.

In der Praxis, d. h. beim Test im Fahrzeug, ist das Einstellen der Bedingungen für die Messungen B2 und B3 meist nicht möglich, da:

1. die technischen Randbedingungen nicht gegeben sind und
2. die funktionale Sicherheit [81] eines sicherheitsbezogen Systems, wie dem Motorsteuergerät, nicht gewährleistet werden kann.

Im ersten Fall sind der Zugriff auf die notwendigen Betriebssystemfunktionen zur Unterdrückung von Interrupts und der Zugriff auf die Register zum Deaktivieren des Caches eines im Fahrzeug verbauten Steuergerätes äußerst schwierig. Im zweiten Fall können bei unterdrückten Interrupts notwendige Funktionen nicht ausgeführt werden, so dass die korrekte Funktion des Steuergerätes nicht gegeben ist. Bei deaktiviertem Cache werden die Laufzeiten der Fahrzeugfunktionen erhöht, so dass ebenfalls die korrekte Funktion nicht gegeben ist.

Die Messungen B1 bis B3 sind daher am HiL-Simulator durchgeführt worden. Die funktionalen Testfälle sind manuell über die im Abschn. 4.4.3 beschriebene Testumgebung eingegeben worden.

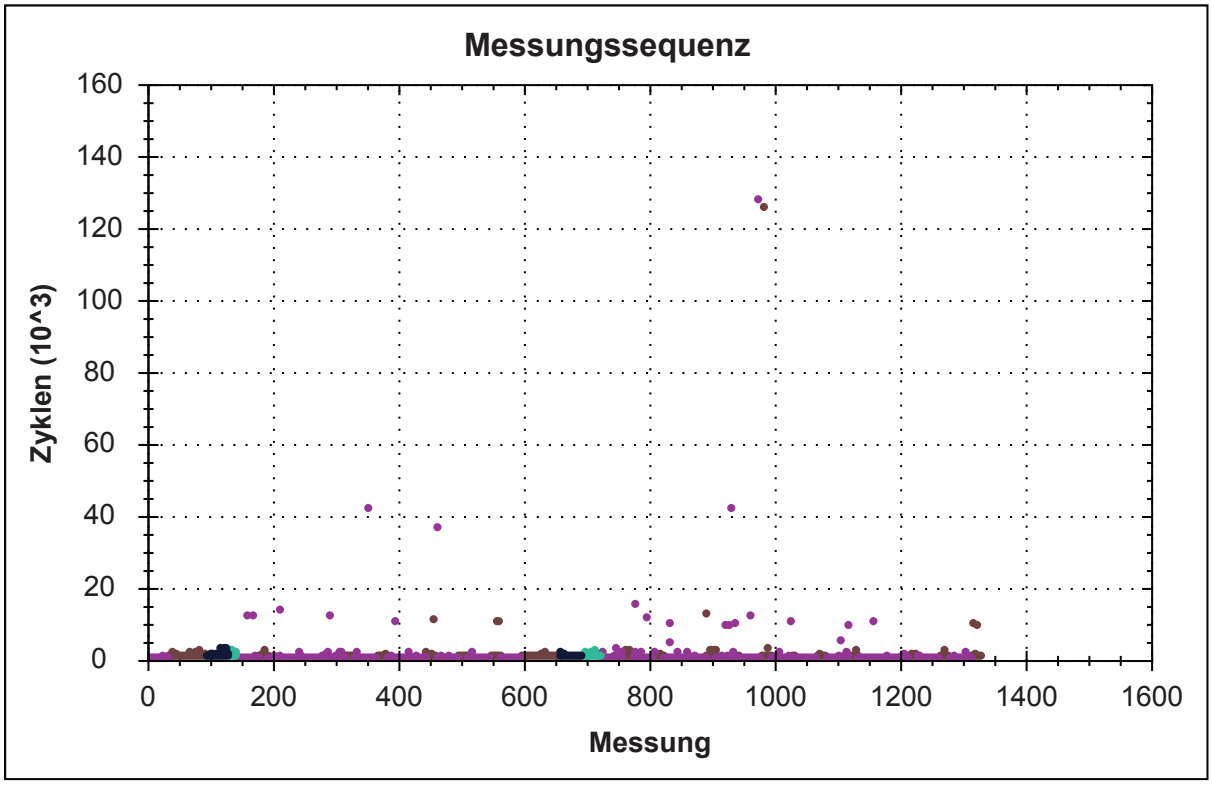


Abb. 6.7: Laufzeitmessung B1 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1}

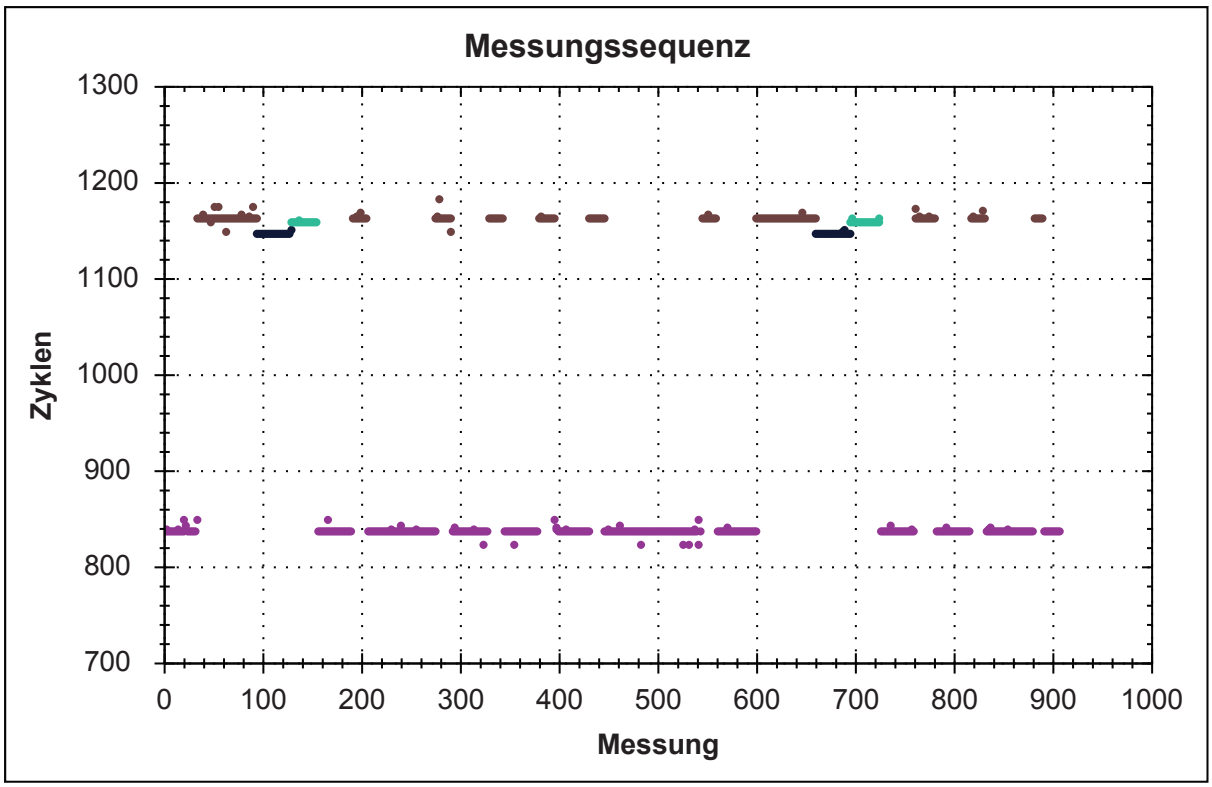


Abb. 6.8: Laufzeitmessung B2 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1} ohne Interrupt

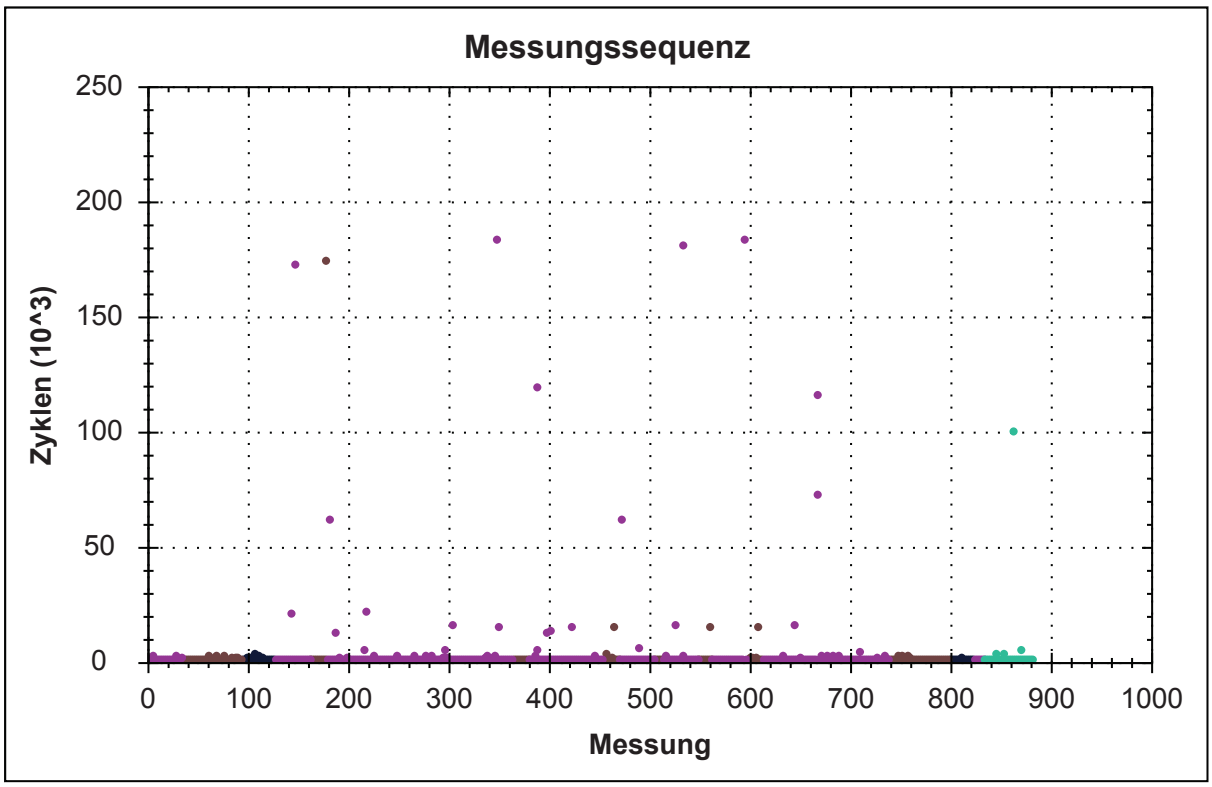


Abb. 6.9: Laufzeitmessung B3 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1} ohne Cache

Tabelle 6.10: Laufzeitmessung A2 der Funktion PR_TimMdl_10ms() bei n=0min^{-1}

ID	Signatur	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
2	/t1/z1/z2/z1001/z1003/z1005/z5/z1001/z1003/z1005/t2		10697	1158	2579
5	/t1/z1/z3/z5/z1001/z1003/z1004/t2		10391	1146	2674
7	/t1/z1/z2/z1001/z1002/z4/t2		10918	823	1052
9	/t1/z1/z2/z1001/z1003/z1004/z5/z1001/z1002/t2		12231	1148	2161

a Darstellung in Abb. 5.10, b Timerzyklen *c*

Tabelle 6.11: Laufzeitmessung B1 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1}

ID	Signatur	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
2	/t1/z1/z2/z1001/z1003/z1005/z5/z1001/z1003/z1005/t2		2645	1158	1418
5	/t1/z1/z3/z5/z1001/z1003/z1004/t2		3180	1129	1305
7	/t1/z1/z2/z1001/z1002/z4/t2		127769	823	1374
9	/t1/z1/z2/z1001/z1003/z1004/z5/z1001/z1002/t2		125860	1148	1760

a Darstellung in Abb. 6.7, b Timerzyklen *c*

Tabelle 6.12: Laufzeitmessung B2 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1} ohne Interrupt

ID	Signatur	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
2	/t1/z1/z2/z1001/z1003/z1005/z5/z1001/z1003/z1005/t2		1163	1158	1158
5	/t1/z1/z3/z5/z1001/z1003/z1004/t2		1150	1146	1146
7	/t1/z1/z2/z1001/z1002/z4/t2		848	823	836
9	/t1/z1/z2/z1001/z1003/z1004/z5/z1001/z1002/t2		1182	1148	1162

a Darstellung in Abb. 6.8, b Timerzyklen *c*

Tabelle 6.13: Laufzeitmessung B3 der Funktion PR_TimMdl_10ms() bei n=4500min^{-1} ohne Cache

ID	Signatur	Farbe[a]	WCET[b]	BCET[b]	Durchschnitt[b]
2	/t1/z1/z2/z1001/z1003/z1005/z5/z1001/z1003/z1005/t2		100405	1342	3570
5	/t1/z1/z3/z5/z1001/z1003/z1004/t2		3260	1280	1396
7	/t1/z1/z2/z1001/z1002/z4/t2		183282	914	3309
9	/t1/z1/z2/z1001/z1003/z1004/z5/z1001/z1002/t2		174067	1316	2437

a Darstellung in Abb. 6.9, b Timerzyklen *c*

Tabelle 6.14: Laufzeitmessung A2, B1, B2, B3 der Funktion PR_TimMdl_10ms()

	A2	B1	B2	B3
BCET[a]	823(ID 7)	823(ID 7)	823(ID 7)	914(ID 7)
WCET[a]	12231(ID 9)	127769(ID 7)	1182(ID 9)	183282(ID 7)
maxBCET[a]	1158(ID 2)	1158(ID 2)	1158(ID 2)	1342(ID 2)

a Timerzyklen *c*

Die Tabellen 6.10 bis 6.13 zeigen die WCET, BCET und Durchschnittslaufzeiten der einzelnen Pfade bei unterschiedlichen Bedingungen. In Tabelle 6.14 werden die Ergebnisse der einzelnen Messung zusammengefasst. Die Betrachtung der BCET führt in den Messungen A2, B1 und B2 zum gleichen Ergebnis, unabhängig von den Einflüssen der anderen Tasks. Dagegen führt die Bestimmung der WCET aufgrund von Cache- und Multitasking-Effekten jeweils zu unterschiedlichen Laufzeiten und Pfaden.

Der Ansatz nach Gl. 6.10 betrachtet nur die untere Pfadlaufzeitgrenze (BCET des Pfades). Aus diesen Laufzeiten wird nun der Pfad mit der größten BCET ausgewählt und als längster gemessener Pfad beschrieben. Die Messungen A2, B1 und B2 kommen so zum gleichen Messergebnis und dem gleichen Programmpfad.

Ansatz zur Bestimmung der WCET bei zufälligen Abweichungen: Für zufällige Abweichungen kann die korrigierte maximale Messobjektlaufzeit bei vollständiger Programmpfadabdeckung wie folgt bestimmt werden.

$$\Delta t_{\mathrm{Pr}\,WCET} = \max\left(\Delta t_{\mathrm{Pr}\,BCET}^{Pfad-ID}\right) \tag{6.11}$$

Zur Beurteilung des so ermittelten Ergebnisses wird die Vorgehensweise zur Ermittlung des Wertes von $\Delta t_{\mathrm{Pr}\,WCET}$ mit angegeben, d. h., hier wird durch Untersuchung der pfadabhängigen BCET auf den längsten Pfad geschlossen $\Delta t_{\mathrm{Pr}\,\max BCET}$.

Die Messung B3 macht auf eine weitere Bedingung aufmerksam, unter der das Messobjekt betrachtet werden muss. In B3 werden nicht die Interrupts deaktiviert, sondern das Programm ohne Cache ausgeführt. Die bisherige Betrachtung der unteren Laufzeitgrenze führt dazu, dass nur Pfadlaufzeiten betrachtet wurden, wenn der Cache zu einer hohen Beschleunigung der Pfadlaufzeit geführt hat. Real sind jedoch Zustände zu erwarten, bei denen der Cache keine Befehle oder Daten enthält, wie zum Beispiel nach dem Reset des Steuergerätes. Für eine sichere Abschätzung der WCET ist es daher erforderlich, das Messobjekt auch unter dieser Bedingung zu beobachten. Den Cache vor einer Messung zu leeren, wie es in [56] vorgeschlagen wird, ist meist bei eingebetteten Systemen mit sehr hohem Aufwand verbunden oder unter realen Bedingungen, wie bei einer Fahrzeugmessung, nicht möglich, da hierfür entsprechende Register zur Konfiguration des Caches nicht zugänglich sind oder der Cache mit ungültigen Daten belegt werden muss. Eine andere Möglichkeit besteht in der Ausführung der Software-Funktion aus einem ungecachten Speicherbereich des Steuergerätes. Dies führt jedoch zwangsläufig zu einer Überschätzung der tatsächlichen Laufzeit, da hier die Beschleunigung durch den Cache bei Mehrfachaufrufen von Unterfunktionen unberücksichtigt bleibt. Die Tabelle 6.13 betrachtet eine Messung ohne Cache und führt ebenfalls bei $\Delta t_{\mathrm{Pr}\,\max BCET}$ zum Pfad mit der ID 2.

6.5 Fazit

Unter der Voraussetzung, dass die Maßnahmen zur Reduktion der unbekannten systematischen Messgeräteabweichung erfüllt werden, ist eine Laufzeitmessung mit einer ähnlich hohen Genauigkeit wie der des LSA, jedoch ohne dessen Nachteile (Zugänglichkeit von Bussignalen), zu erreichen. Mit dem Ansatz aus Gl. 6.11 wird eine mathematische Analyse von Laufzeitmessreihen möglich, auch wenn die zu messende Software-Funktion durch zufällige Messabweichungen überlagert ist. Über Experimente und Analysen konnte festgestellt werden, dass die Verteilungsfunktionen die meisten auftretenden Verteilungen in praktisch aufgenommenen Messreihen beschreiben können. Diese sind in Abb. 6.7, 6.8 und 6.9 aufgeführt. Dazu wird für die jeweilige Verteilung noch angegeben, was für sie charakteristisch ist (z. B. Fahrzeugmessung) und bei welchen typischen Vorgängen sie angenommen werden kann (z. B. ohne Cache, ohne Interrupts). Hier wird von statistisch ausreichend großen Mengen ausgegangen, es können jedoch auch nur Pfade einmalig oder nur wenige Male ausgeführt sein. Sind diese Pfade von zufälligen Abweichungen überlagert, führt das unweigerlich zu einer Überschätzung der Pfadlaufzeit. Folglich treffen erzielte Aussagen zu den pfadabhängigen Laufzeiten dann nicht in dem gewünschten Maße zu. Die Sicherheit des Systems wird durch diese Überschätzung nicht beeinflusst, da ein Überschreiten der Prozess-Containergrenze zu weiteren Analysen des Programmpfades führen würde. Eine Überschätzung der Pfadlaufzeit innerhalb der Containerlaufzeit führt zu ungenutzten Ressourcen bzw. unnötig vorgehaltenen Ressourcen. Um dies zu verhindern, sollte das Histogramm zu dem ermittelten WCET-Pfad betrachtet werden. Hier sollte immer eine Rechtsschiefe beobachtet werden, da die untere Laufzeitgrenze architekturbedingt nicht unterschritten werden kann (s. Abb. 6.10). In der Praxis hat sich hier gezeigt, dass zur Beurteilung dieses WCET-Pfades die Häufigkeit des Wertes von $\Delta t_{\mathrm{Pr}\,\max BCET}$ mit angegeben wird, z. B. reicht hier eine Anzahl von Einzelmesswerten von

>10. Wie diese Unsicherheit zu behandeln ist oder ob sie vernachlässigt werden darf, muss der Messtechniker auf der Basis seiner Kenntnisse der vorliegenden Messauswertung entscheiden. Zudem können auch zwei Testfälle für den Serienintegrationstest beim Systemlieferanten angegeben werden, einer der unter realen Bedingungen (mit Cache) und einer der unter Versuchsbedingungen (ohne Cache) die obere Laufzeitgrenze stimuliert. Es ist allerdings zu beachten, dass unter Umständen durch diese Vorgehensweise eine unerwünscht konservative Auslegung der Prozesslaufzeit bewirkt wird. Folglich bleiben Ressourcen ungenutzt.

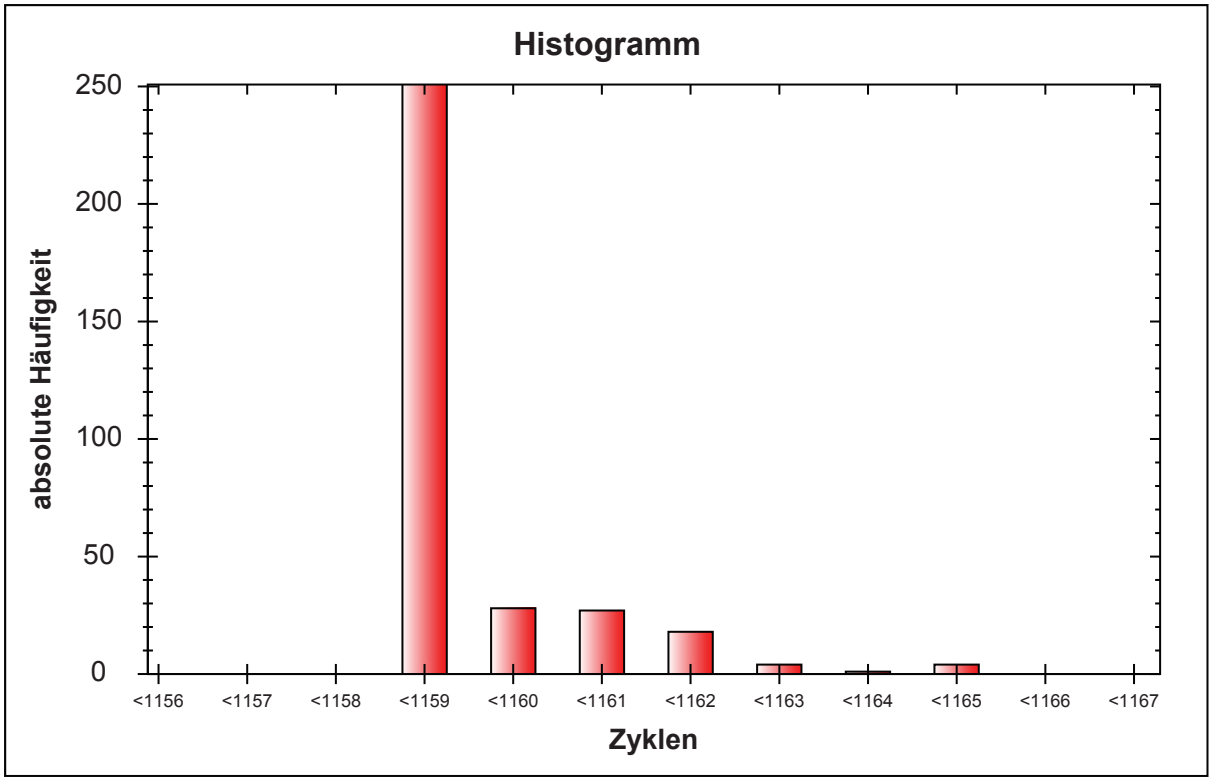


Abb. 6.10: Diskrete Messwertverteilung Pfad-ID 2 der Funktion PR_TimMdl_10ms()

7 Funktionspfadorientierte Datenanalyse

Ziel der funktionspfadorientierten Datenanalyse ist es, die Nachteile der entwickelten programmpfadorientierten Datenanalyse bezüglich der fehlenden Pfadabdeckung[1] auszugleichen. Dies soll unter Ausnutzung der Software-Eigenschaften von modellbasierten Fahrzeugfunktionen erfolgen. Zunächst werden diese Software-Eigenschaften analysiert und beschrieben. Hieraus werden zusätzliche Regeln für den temporalen Test abgeleitet und notwendige Vorgehensschritte eingeführt. Die funktionspfadabhängige Laufzeitbestimmung stellt den abschließenden Verfahrensschritt der pfadabhängigen Laufzeitbestimmung dar und damit eine Methode bereit, um Schlüsse mittels eines statischen Laufzeitmodells einer Fahrzeugfunktion ziehen zu können. Mit diesem Laufzeitmodell wird es möglich, die gemessene WCET mit einer berechneten theoretischen WCET zu vergleichen.

7.1 Eigenschaften modellbasierter Fahrzeugfunktionen

Die Funktionsmodelle von Fahrzeugfunktionen basieren auf standardisierten Bibliotheken von Basisblöcken zur Anwendung im Automobilbereich. Basisblöcke sind sowohl einfache Operatoren, wie logische, vergleichende und arithmetische Operatoren, aber auch Funktionen wie Zeitgeber oder regelungstechnische Blöcke wie Tiefpass. Unter der Annahme, dass in verschiedenen Funktionsmodellen identische Basisblöcke vorliegen und sich ein Funktionsmodell ausschließlich aus diesen Basisblöcken zusammensetzt, beschränkt sich ein Modell auf eine Netzliste, welche die Verknüpfung der Basisblöcke definiert. Der folgende Ansatz erfordert die Bereitstellung von Basisblöcken in verschiedenen Funktionsmodellen, die gleiches Verhalten und gleiche Laufzeiten aufweisen. Das heißt, dieser Ansatz ist architektur- und compilergebunden. Um gleiches Verhalten zu garantieren, wird eine Referenzbibliothek [37, 87] genutzt, von der die Laufzeiten bestimmt werden müssen.

7.1.1 Relevanz für die vorliegende Arbeit

Ein Ziel des folgenden Abschnitts ist es, eine Methode bereitzustellen, um die Kenntnisse über automotive Standardfunktionen hinsichtlich des Laufzeitverhaltens zu erweitern, so dass eine Untersuchung jeder einzelnen Instanz dieser grundlegenden Funktionen zukünftig ausgelassen werden kann (diese Aussage gilt nur für die analysierten Bedingungen wie Compiler, Steuergeräte Plattform usw.). Aus diesem Anspruch ergibt sich, dass die zu untersuchenden Funktionen aus einer für den Anwendungsbereich spezifischen Funktionsbeschreibung wie dem ASAM Standard Blockset ausgewählt werden sollen.

7.1.2 ASAM Standard Blockset als Funktionsbeschreibung

Die ASAM MBFS (Model Based Function Specification) Arbeitsgruppe besteht aus einem Gremium der Firmen Audi, Bosch, DaimlerChrysler, dSpace und Siemens. Das Ziel der Arbeitsgruppe ist es, unabhängig vom ausgewählten Modellierungstool eine standardisierte

[1] Risiko von ungetesteten Programmpfaden (s. Kapitel 4)

Funktionsbibliothek für blockbasierte Entwicklungsumgebungen zu spezifizieren, die in automotiven Anwendungen zum Einsatz kommt, mit speziellem Fokus auf Funktionen des Motorsteuergerätes.

Das Blockset beschreibt neben der Funktionalität jeder Funktion einen eindeutigen Namen sowie ein eindeutiges Symbol für die Funktionsbibliothek, welches auf die Funktionalität hinweist. Es werden Eingangs- und Ausgangsschnittstellen, Parameter, Zustände und interne Konstanten beschrieben und Symbole (z. B. *u* für einen Signaleingang) zugewiesen.

Die Implementierung der einzelnen Funktionalität wird nicht genau festgelegt, jedoch mittels eines Pseudocodes für viele Funktionen beschrieben. Es werden Eigenschaften und Verhalten sowohl für die einzelnen Blöcke als auch für das gesamte Blockset definiert. Es findet keine Konsistenzprüfung der Eingangswerte statt. Das bedeutet, dass es Aufgabe der Funktionsentwicklung ist, eine Division durch Null zu vermeiden, da dies nicht durch die Blockimplementierung abgefangen wird. Des weiteren werden Testfälle und Ergebnisse vorgegeben sowie Simulationsergebnisse grafisch dargestellt. Neben der Beschreibung der Funktionsblöcke werden außerdem Festlegungen zur Modellierung von Blöcken getroffen (nach dem Prinzip eines Metamodells), die beispielsweise definieren, wie die Ein- und Ausgänge um einen Funktionsblock angeordnet sind und wo deren Namen platziert werden. Die MBFS-Beschreibung des RSFlipFlop aus der Beispielfunktion PR_TimMdl ist im Anhang A.4 angegeben. Der hier vorgeschlagene Ansatz erweitert die Blockset-Beschreibung um temporale Eigenschaften der jeweiligen Funktion.

7.2 Verfahren zur funktionspfadabhängigen Laufzeitbestimmung

Aus den im vorangegangenen Abschnitt beschriebenen Eigenschaften leiten sich die folgenden Regeln zur funktionspfadabhängigen Laufzeitbestimmung ab. Die zu messende Software-Funktion wird in einzelne Segmente aufgeteilt. Die Segmentgröße ist auf Funktionen begrenzt. Gegenüber der Segmentierung auf Funktionsebene, wie sie hier vorgeschlagen wird, sind andere Segmentierungen gebräuchlich, wie zum Beispiel auf Grundblockebene (s. Abschn. 3.2.5) und Single-Feasible-Path-Ebene (s. Abschn. 3.3 [3]).
Die wesentlichen Unterscheidungsmerkmale sind hierzu:

1. Die Laufzeitinformationen von standardisierten Bibliotheksfunktionen können in unterschiedlichen Funktionsmodellen wiederverwendet werden.
2. Die Laufzeiten von standardisierten Bibliotheksfunktionen können ohne Bezug zu einer Fahrzeugfunktion bestimmt werden.
3. Für Mehrfachaufrufe von standardisierten Bibliotheksfunktionen können Cacheeffekte für den *n*-ten Aufruf berücksichtigt werden.
4. Eine vollständige Pfadabdeckung durch die wenigen Programmpfade der einzelnen standardisierten Bibliotheksfunktionen ist möglich.

Bei der funktionspfadabhängigen Laufzeitbestimmung wird nicht mehr versucht, eine hohe Pfadabdeckung der gesamten Software-Funktion zu erreichen, vielmehr genügt es, die Pfadabdeckung auf Messsegmentebene (Funktionsebene) zu realisieren. Dafür wird im Rahmen dieses Kapitels eine Methode beschrieben, bei der die pfadabhängige Laufzeitmessung um die funktionspfadabhängige Laufzeitmessung erweitert wird. Das bedeutet, dass bei einer pfadabhängigen Laufzeitmessung für den gesamten Prozess einer Software-Funktion auch die Unterfunktionen (Messsegmente) auf ihr Laufzeitverhalten untersucht werden. Mithilfe der so gewonnenen Daten kann im Anschluss die theoretisch mögliche maximale Dauer der Ausführungszeit für den gesamten Prozess rekonstruiert werden. In einem Laufzeitmodell werden dazu die einzelnen Segmentlaufzeiten (Funktionslaufzeiten) addiert. Für die theoreti-

sche Worst-Case-Laufzeit werden die Segmentpfade mit den längsten Ausführungszeiten der Software-Funktion addiert. Dieses Laufzeitmodell stellt eine Möglichkeit dar, den gewählten Testfall zu bewerten, d. h. zu beurteilen, wie repräsentativ der Testfall mit der gemessenen Laufzeit gegenüber der berechneten theoretischen WCET des Prozesses ist.

7.2.1 Hierarchielokale Pfadabdeckung

Nachfolgend wird auf die Mittel des strukturorientierten Tests zurückgegriffen und die hierarchielokale Pfadabdeckung als Testüberdeckungsanforderung eingeführt. Unter hierarchielokaler Pfadabdeckung ist eine vollständige Pfadabdeckung auf einer Hierarchieebene (Funktionen) zu verstehen (s. Abb. 7.1), bei der die Pfadabdeckung auf tieferen Hierarchieebenen unberücksichtigt bleibt. Die hierarchielokale Pfadabdeckung definiert damit die erweiterten Anforderungen an den Software-Modultest[2]. Um den Aufwand für die Versuchsplanung nicht wesentlich zu erhöhen, sollte angestrebt werden, mit den funktionalen Testfällen eine hierarchielokale Pfadabdeckung zu erreichen. Strukturorientierte Tests kommen daher nur für hierarchielokale Pfade infrage, die nicht durch funktionsorientierte Tests erreicht wurden.

Grundsätzlich muss auf jeder Funktionsebene eine vollständige Pfadabdeckung erzielt werden. Für die höhere Funktionsebene ist anzunehmen, dass die untere aufgerufene Ebene nur aus einem Pfad besteht. Durch diese Maßnahme soll eine hierarchielokale Pfadabdeckung bei höheren Ebenen erreicht werden.

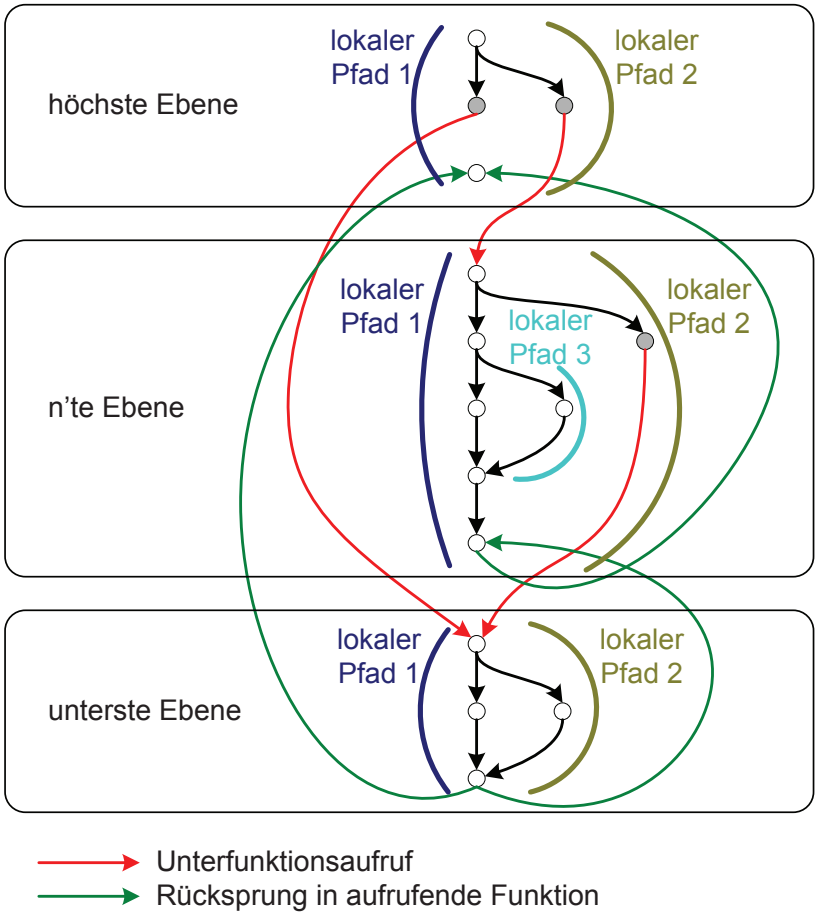


Abb. 7.1: Pfadabdeckung auf Hierarchieebene

[2] Die hierarchielokale Pfadabdeckung ist durch die bestehenden Anforderungen an den Software-Modultest nicht abgedeckt (minimale Anforderung: Zweigabdeckung, s. Abschn. 4.2.1).

Zum Stimulieren der unteren Funktionsebenen ist es notwendig, dass zunächst die höher liegenden Ebenen mit entsprechenden Testfällen stimuliert werden. Die Testfälle müssen so aufgebaut werden, dass die notwendigen Zustände der unteren Ebenen erreicht werden können. Dies erfordert eine sehr genaue Analyse aller möglichen Eingabebedingungen, da die unterste Ebene nicht bzw. nur sehr selten separat stimulierbar ist. Die unterste Funktionsebene wird meist von standardisierten Bibliotheksfunktionen belegt, die in einer Systembibliothek zusammengefasst sind. Durch diese Systembibliothek wird folgende Vereinfachung möglich.

Systembibliothek Den Systembibliotheksfunktionen wird in dieser Arbeit eine besondere Stellung eingeräumt. Da diese sowohl laufzeit- als auch speicheroptimiert sind, sind deren Laufzeiten und Pfade separat nach dem im Kapitel 4 entwickelten Verfahren erfasst. Die Daten sind in der Laufzeitdatenbank abgelegt und können dort abgefragt werden. Das vereinfacht die Laufzeitmessung auf den Funktionsebenen. Hierfür reicht es aus, wenn die aufgerufene Bibliotheksroutine entsprechend den diskutierten Vorgaben auf einem Pfad durchlaufen wird. Neue Laufzeitmessungen sind nur bei Bibliothekswechsel oder Architekturwechsel notwendig. Durch diese Maßnahmen reduziert sich der zusätzliche Aufwand für die hierarchielokale Pfadabdeckung im Software-Modultest.

7.2.2 Laufzeitinstrumentierung

Die Instrumentierung des Programmcodes für die Laufzeitmessung erfolgt nach dem im Abschn. 4.4.1 beschriebenen Messprinzip. Dazu wird der zu messende Programmcode mit Messpunkten instrumentiert (s. Abb. 7.2). Die Messpunkte werden am Anfang und am Ende des zu messenden Abschnittes gesetzt (hier mit $t_{1,1},\ldots,t_{n,2}$ für die Funktionen 1 bis n und t_1, t_2 für den Prozess bezeichnet). Die Laufzeit dieses Abschnittes ergibt sich dann aus der Differenz der beiden Messwerte ($t_{k,1}$, $t_{k,2}$). Bei der Laufzeitmessung wird die Laufzeit auf Funktionsebene gemessen, d. h., direkt nach Funktionseintritt und unmittelbar vor dem Funktionsaustritt wird ein Messpunkt im Programmcode platziert. Ebenso ist es auch möglich, mit der benannten Technik auf Segment-, Block- oder Anweisungsebene die Laufzeit zu messen.

Wie bereits im Kapitel 6 eingeführt, besteht ein Messpunkt aus einer Messpunktnummer bzw. Messpunkt-ID und dem Messwert. Diese beiden Werte werden an einem fortlaufenden Index (*MP_INDEX*) in den Vektor *tmpVWMP_GEN* geschrieben. Um die gemessene Laufzeit mit dem dazu durchlaufenen Pfad zu kombinieren (s. Kapitel 5), werden die durchlaufenen Messpunkt-IDs zusätzlich zu den Zweig-IDs in der Reihenfolge ihres Auftretens in den Vektor zur Pfadverfolgung (*tmpVWPATH_GEN*) eingetragen. Dafür sorgt der fortlaufende Index *PATH_INDEX*. Die Messpunkt- und Zweig-ID's müssen eindeutig sein, da diese im Vektor zur Pfadverfolgung (*tmpVWPATH_GEN*) in beliebiger Folge abgelegt werden. Die Abb. 7.3 zeigt den generierten Funktionscode des Blockset-Elementes RSFlipFlop der standardisierten Bibliotheksfunktion mit der Instrumentierung für Laufzeitmessung und Pfadverfolgung.

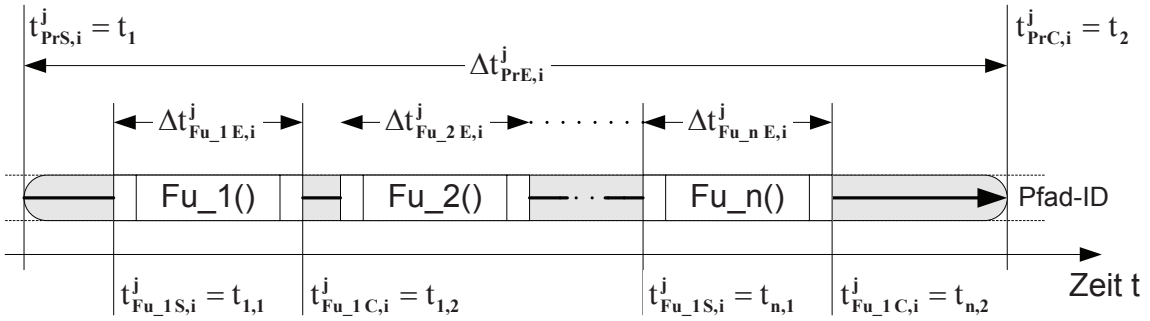


Abb. 7.2: Laufzeitmesspunkte für Prozess mit Unterfunktionen 1 bis n

Zu beachten ist, dass sich die Instrumentierungen am Funktionsbeginn und am Funktionsende durch die fehlende Codezeile *PATH_POINT(Zweig-ID)* unterscheiden, die am Ende fehlt. Solange keine Verzweigung auftritt, ist der durchschrittene Programmpfad eindeutig und durch den vorherigen *PATH_POINT(Zweig-ID)* bestimmt. Die einzige Ausnahme ist der Funktionsbeginn, bei dem festgelegt werden muss, wie der Initialzweig der jeweiligen Funktion bezeichnet werden soll. Die Implementierung der Laufzeitinstrumentierung in den ASCET-Codegenerator wird in [33] beschrieben.

```
/*********************************************************
Component:      RSFlipFlop
Implementation: U8
Generated by:   ASCET-SD V5.1.3
ASCET-SE:       TriCore [V5.2.0] RT_ANALYSIS
*********************************************************/

#include <STD_TYPE.H>
#include "asd_patch.h"


#define DO_RT_ANALYSIS
#define DO_PATH_ANALYSIS


#include "rt_analysis.h"
#ifndef _RSFLIPFLOP_RT_H
#include "rsflipflop_rt.h"
#endif /* #ifndef _RSFLIPFLOP_RT_H */

/*** a multiple instance class ***/

#define _status self->status

/* define method RSFLIPFLOP_RT_U8_compute (modelled as
compute_RSFLIPFLOP_RT_U8) */
/*** functions called by method RSFLIPFLOP_RT_U8_compute ***/

/* begin of method RSFLIPFLOP_RT_U8_compute (modelled as
compute_RSFLIPFLOP_RT_U8) */
void RSFLIPFLOP_RT_U8_compute (struct RSFLIPFLOP_RT_U8 *self, uint8 r,
   uint8 s)
{
        #ifdef DO_RT_ANALYSIS
        /* Generated TIME INSTRUMENT */
        tmpVWMP_GEN[MP_INDEX++] = 10001;
        tmpVWMP_GEN[MP_INDEX++] = (*(volatile uint32*)0xF0000210);
        #ifdef DO_PATH_ANALYSIS
        tmpVWPATH_GEN[PATH_INDEX++] = 10001;
        PATH_POINT(1001);
        #endif // DO_PATH_ANALYSIS
        #endif // DO_RT_ANALYSIS
        if (PATH_COND(1002, 1003, (int)(r)))
        {
                (self->status = (uint8)false);
        }
        else
        {
                if (PATH_COND(1004, 1005, (int)(s)))
                {
                        (self->status = (uint8)true);
                } /* end if */
        } /* end if */
        #ifdef DO_RT_ANALYSIS
        /* Generated TIME INSTRUMENT */
        tmpVWMP_GEN[MP_INDEX++] = 10002;
        tmpVWMP_GEN[MP_INDEX++] = (*(volatile uint32*)0xF0000210);
        #ifdef DO_PATH_ANALYSIS
        tmpVWPATH_GEN[PATH_INDEX++] = 10002;
        #endif // DO_PATH_ANALYSIS
        #endif // DO_RT_ANALYSIS
}
/* end of method RSFLIPFLOP_RT_U8_compute (modelled as
compute_RSFLIPFLOP_RT_U8) */
```

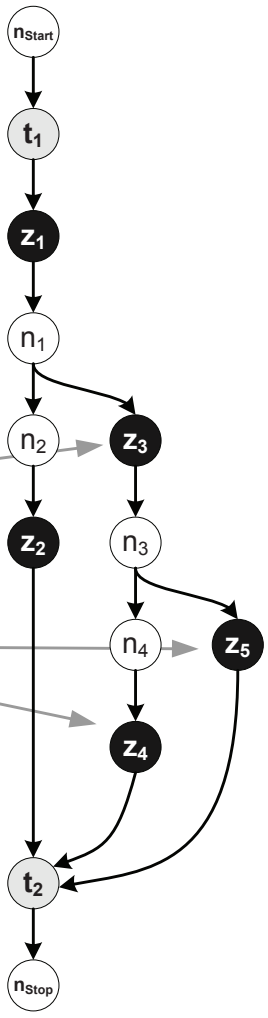


Abb. 7.3: Beispielfunktion mit Instrumentierung

7.2.3 Erhebungsschema

Am Ende eines Tasks werden die Eingangs- und Ausgangssignale sowie der Vektor *tmpVWPATH_GEN* oder der Vektor *tmpVWMP_GEN* aus dem Speicher des Steuergeräts übertragen. Die Zuordnung von Mess- und Eingangsdaten ist durch den Messablauf gegeben, bei jedem Auslesen des Steuergerätespeichers werden die Werte dem Messprotokoll hinzugefügt (s. Kapitel 4). Die Vektoren mit den Messdaten werden vor der nächsten Ausführung des Messprozesses gelöscht und anschließend beginnend vom ersten Vektor-Element wieder beschrieben.

Um die Laufzeit des Messprozesses nicht mit zusätzlichem Mess-Overhead durch das Löschen des Vektors und das Übertragen der Messdaten zum Monitorsystem zu beaufschlagen, findet die Initialisierung der Messgrößen jeweils vor der Messung zu Beginn des Tasks oder im Schnittstellenprozess (A1) statt. Die Übertragung der Messgrößen wird hingegen nach der Messung durch einen Systemprozess außerhalb des zu messenden Prozess durchgeführt (s. Abb. 7.4).

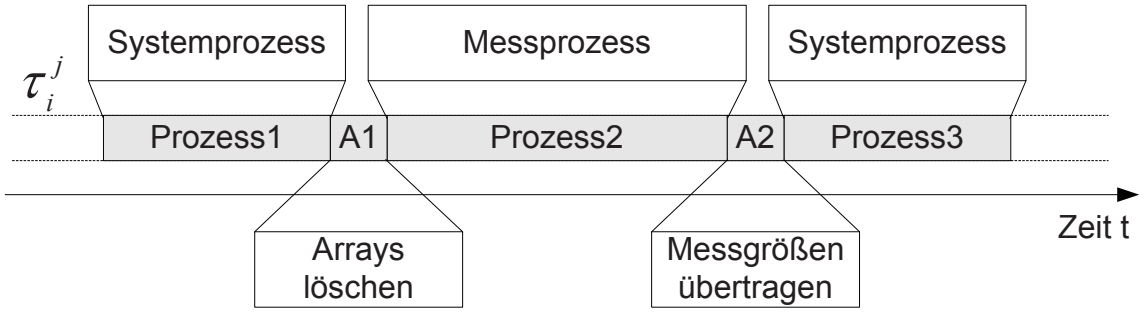


Abb. 7.4: Messprozess im Task *i* bei der *j*-ten Ausführung (τ_i^j)

Werden die Eingangsgrößen verändert, kann sich auch der Pfad ändern. Abhängig vom Pfad ändert sich so auch die Anzahl der durchlaufen Zweige und aufgerufenen Funktionen. Der Vektor für das Zwischenspeichern der Messpunkte muss so groß sein, dass die maximale Anzahl an Messpunkten der Prozesspfade aufgenommen werden kann. Das Gleiche gilt für den Vektor, der die Signatur eines Prozesspfades aufnehmen soll. Die maximale Anzahl ist in beiden Fällen durch die vorgegebenen Software-Metriken des Qualitätsmodells begrenzt (s. Abschn. 4.2.1), die Prüfung erfolgt im Software-Codetest.

7.3 Analyse funktionspfadabhängiger Laufzeitmessdaten

Die statische Auswertung der pfadabhängigen Laufzeitmessdaten soll um eine statische Auswertung der funktionspfadabhängigen Laufzeitmessdaten ergänzt werden. Die Probleme bei der funktionspfadabhängigen Laufzeitanalyse bestehen:

1. in der Zuordnung der Messpunkte zu einer Funktion und
2. in der Zuordnung der Laufzeiten zu den Funktionspfaden.

Zwar kann anhand der Messdaten, welche in der Datenbank organisiert sind, die Laufzeitdifferenz zwischen zwei Messpunkten bestimmt werden, unbekannt bleibt jedoch, welcher Pfad dabei in der Funktion durchlaufen wurde und zu welcher Funktion diese Messpunkte gehören.

In Abb. 7.5 wird das Problem der unbekannten Pfade einer Funktion verdeutlicht. Es wird angenommen, dass bei der Laufzeitmessung einer Software-Funktion (main-Funktion) ein Pfad durchlaufen wurde, in dem eine Unterfunktion (f1) zweimal aufgerufen wird. Dabei wird

beim ersten Funktionsaufruf der Unterfunktionspfad 1 und beim zweiten Funktionsaufruf der Pfad 2 durchlaufen. Durch eine Abfrage der Messdaten aus der Datenbank kann anhand der Start- und Stoppmesspunkte ($t_{1,1}$, $t_{1,2}$) die Laufzeitdifferenz bestimmt werden, als Ergebnis werden hier drei Laufzeiten Δt_x, Δt_y und Δt_z ausgegeben. Die Pfade, die dabei durchlaufen werden, bleiben unbekannt.

Um das Problem zu lösen, werden die durchlaufenen Pfade der main-Funktion, die bei der Pfadverfolgung aufgezeichnet und in der Datenbank abgespeichert sind, herangezogen.

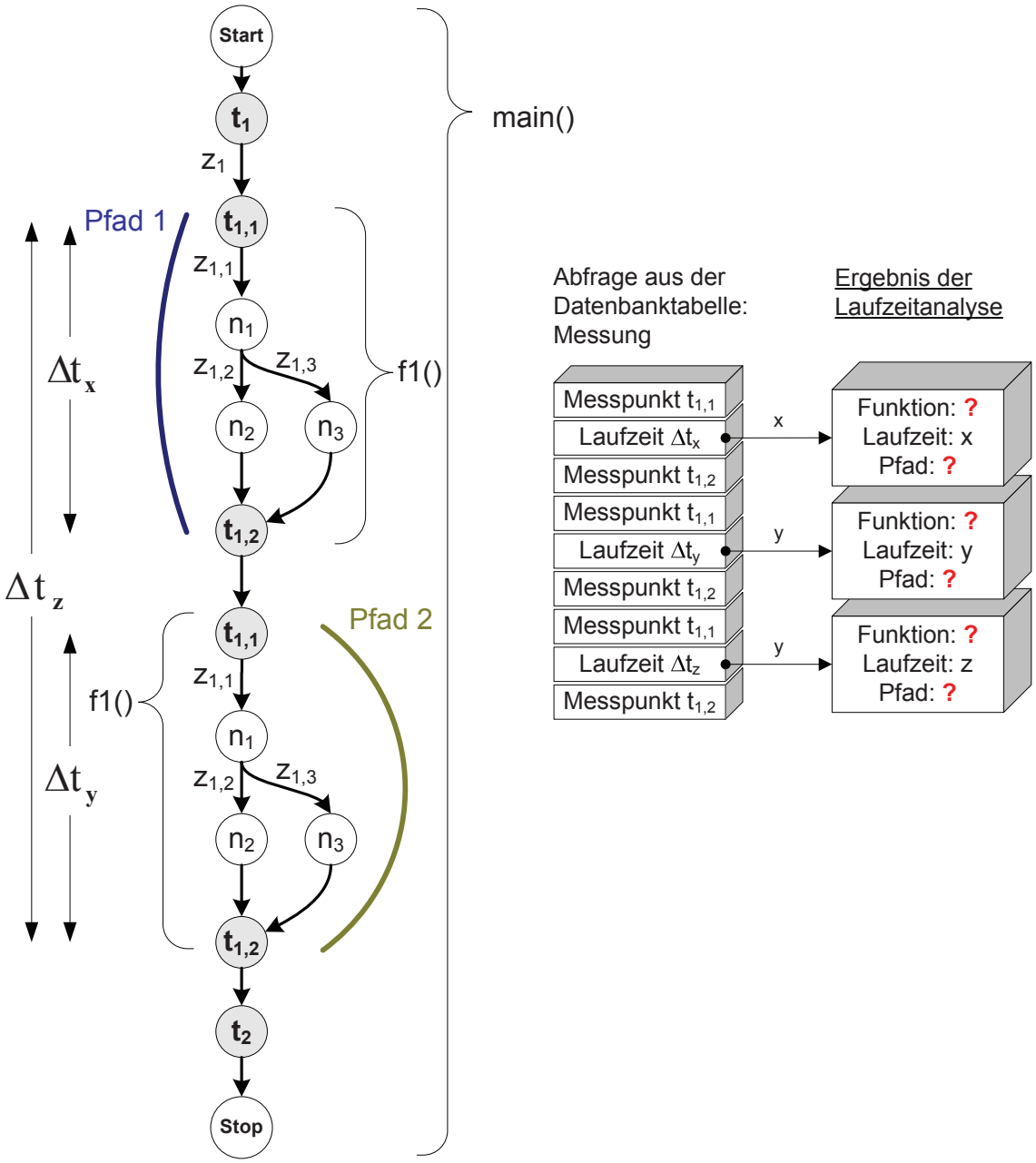


Abb. 7.5: Laufzeitbestimmung bei Unterfunktionen

Anhand der kombinierten Information können die Laufzeiten mit den dazugehörigen Teilpfaden bestimmt werden. Für die Zuordnung von Messpunkten und Teilpfaden zu einer Unterfunktion muss, um eine Wiederverwendung von Laufzeitinformationen von einmal analysierten Funktionen zu ermöglichen, eine Abbildung von Laufzeitinformationen auf die zugehörige Funktion erfolgen. Das heißt, von jeder Funktion müssen statische Merkmale wie

der Funktionsname, die eindeutigen Messpunktnummern und die eindeutigen Pfade (Zweig-ID's) bekannt sein. Um dies leisten zu können, müssen Strukturinformationen der Software-Funktion und deren Unterfunktionen bei der Laufzeitanalyse vorhanden sein. Abbildung 7.6 verdeutlicht die beschriebenen Abhängigkeiten zur Bestimmung der Funktionslaufzeiten und der Funktionspfade einer Unterfunktion.

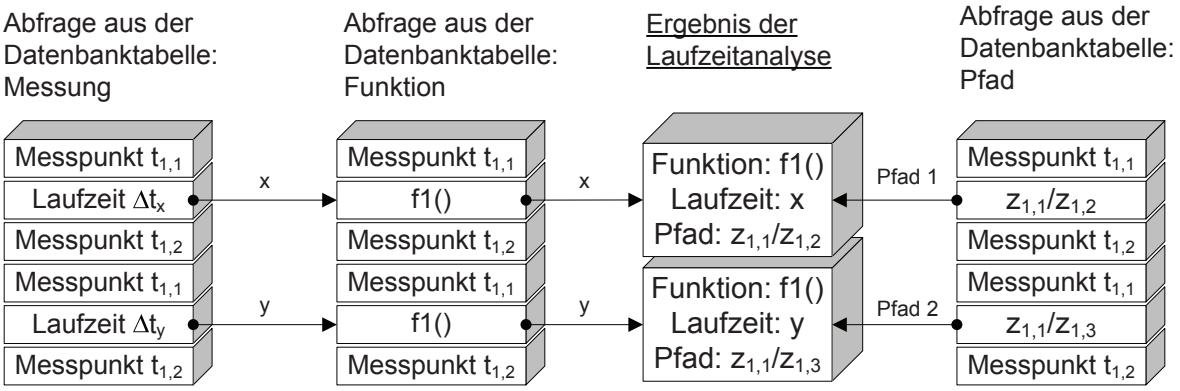


Abb. 7.6: Laufzeit- und Pfadbestimmung einer Unterfunktion

Aus den Messdaten kann zusätzlich ermittelt werden, in welcher Reihenfolge (Position im Pfad der main-Funktion) die gesuchten (Teil-)Pfade innerhalb einer Messung durchlaufen wurden. Die Reihenfolge der Durchläufe ist für die graphische Darstellung des Verlaufs aller Messungen und für die Beurteilung von Cache-Effekten bei Mehrfach-Funktionsaufrufen notwendig. Im Folgenden werden die dafür notwendigen Erweiterungen zur Datengewinnung, Datenorganisation und Datenanalyse beschrieben.

7.3.1 Statische Codeanalyse und Software-Visualisierung

Die Aufgabe der statischen Codeanalyse ist die Darstellung der Software-Struktur in einer interpretierbaren Form und die Bestimmung von statischen Codemerkmalen, wie die Anzahl von möglichen Programmpfaden. Die Software-Visualisierung [89] dient zur Darstellung von Programmpfad und Zweigen. Neue Anforderungen werden durch das hier vorgestellte Prüfkonzept gestellt, in dem auch Zeitmesspunkte aus dem instrumentierten C-Code dargestellt werden müssen.

Die verschiedenen Möglichkeiten zur Programmpfadbeschreibung bzw. Darstellungsmethoden von Programmpfaden sind detailliert in [2] beschrieben. Hierbei werden im Wesentlichen zwei Verfahren zur Programmpfadanalyse herausgestellt. Das ist zum einen die explizite Programmpfadbeschreibung [21, 22] und zum anderen die implizite Programmpfadaufzählung [1]. Diese Verfahren zeichnen sich dadurch aus, dass sie von der Genauigkeit der Benutzervorgaben abhängig sind und bei entsprechend komplexen Programmen mit hohem Aufwand oder nur unvollständig zu realisieren sind.

Für die statische Codeanalyse sollte ein Werkzeug benutzt werden, das in der Lage ist, den Kontrollfluss sowie die vorgenommenen Instrumentierungen graphisch darzustellen und eine Ausgabe der Programmpfadinformationen für die automatisierte Analyse ermöglichen.

Das Ziel der hier durchgeführten statischen Codeanalyse besteht darin, die Messsegmente (Unterfunktionen) für die Laufzeitmessung festzulegen. Durch die statische Codeanalyse wird der Kontrollflussgraph der einzelnen Funktionen/Unterfunktion analysiert sowie alle möglichen Pfade dieser Funktionen bestimmt. Abbildung 7.7 zeigt einen Kontrollflussgraphen einer Software-Funktion bestehend aus vier Pfaden.

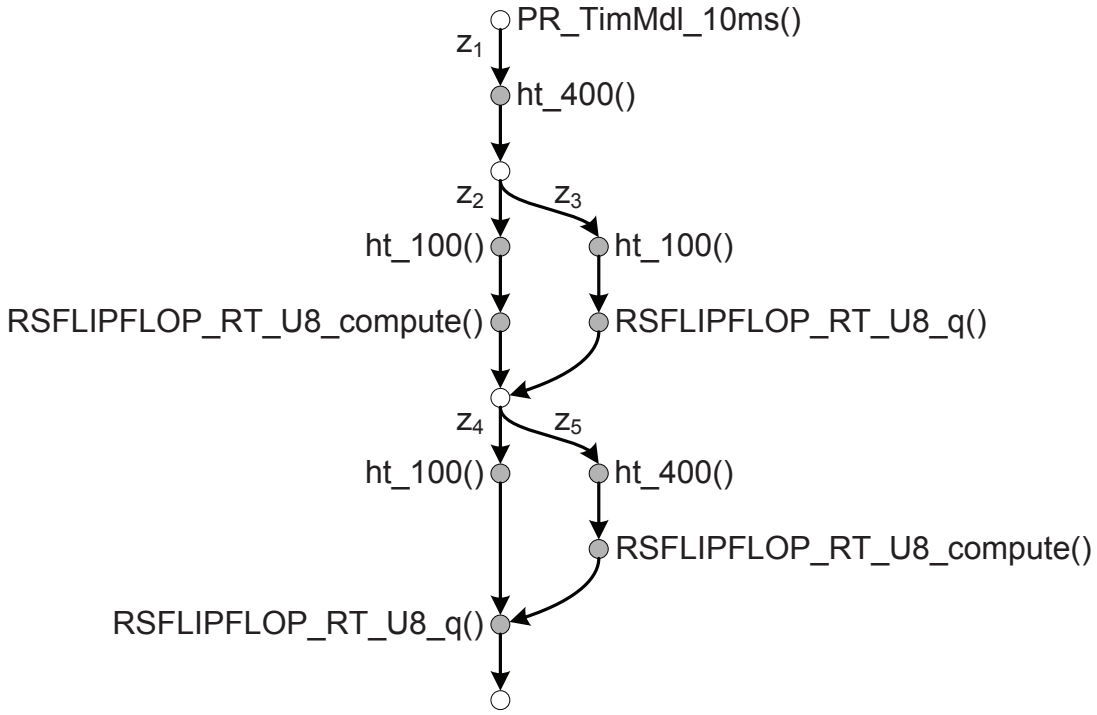


Abb. 7.7: Kontrollflussgraph der Funktion PR_TimMdl_10ms() mit Unterfunktionsaufrufen

Die statische Codeanalyse wird hier anhand der Beispielfunktion aus Kapitel 5 durchgeführt, es werden die Unterfunktionen (s. Abb. 7.7) sowie die Software-Funktion (PR_TimMdl) auf ihren Kontrollfluss untersucht. In der Tabelle 7.1 werden alle Messsegmente aufgelistet.

Tabelle 7.1: Messsegmente Funktion PR_TimMdl_10ms

Funktionsname	mögliche Pfade	Beschreibung
RSFLIPFLOP_RT_U8_compute	3	Funktion (Systembibliothek)
RSFLIPFLOP_RT_U8_q	1	Funktion (Systembibliothek)
ht_100	1	Funktion
ht_400	1	Funktion
PR_TimMdl_10ms	4	main-Funktion (Prozess)

Um den Steuergeräteressourcen (Speicher, Rechenzeit und Übertragungskapazität des Monitorsystems) gerecht zu werden, können Vereinfachungen für die Messung und Analyse der Software-Funktionen festgelegt werden. Für Unterfunktionen, die nur aus einem Pfad bestehen, können die Single-Feasible-Path-Eigenschaften (SFP) nach [2, 3] über die Funktionsebenen hinaus genutzt werden, um die Anzahl der Instrumentierungspunkte zu reduzieren. Dies bringt folgende Vorteile:

1. eine Verringerung der Messunsicherheit durch unbekannte systematische und zufällige Messgeräteabweichungen (s. Kapitel 6) und
2. eine Reduzierung der Datenmenge, die im Steuergerät zwischengespeichert und vom Monitorsystem übertragen werden muss (s. Kapitel 4).

Die nicht instrumentierten Unterfunktionen werden mit ihrem Laufzeitbeitrag der nächsthöheren Hierarchieebene zugerechnet. Bei Einzelausführungspfaden bringt das keinen Nachteil, solange die Laufzeitinformation dieser Funktionen nicht anderweitig verwendet werden soll. Zudem wird der Aufwand für eine hierarchielokale Pfadabdeckung nicht erhöht.

Um eine Wiederverwendung von Laufzeitinformationen in der Datenanalyse zu ermöglichen, müssen die statischen Programminformationen (Funktionsname, Pfad, usw.) mit den dynamischen Programminformationen (Signatur und Laufzeit) kombiniert werden. Dazu ist es notwendig, die Ergebnisse der statischen Codeanalyse in der Datenbank zu organisieren. Zusätzlich ist es notwendig, die Position von Funktionen im Programmpfad zur Berücksichtigung von mehrfachen Funktionsaufrufen aus der statischen Codeanalyse zu ermitteln. Anschließend findet eine Datenfusion zwischen gemessenen und analysierten Pfaden statt. Diese Vorgänge werden in den folgenden Abschnitten näher erläutert.

7.3.2 Messdatenorganisation, Datenfusion und Struktursuche

Die Datenanalyse muss so erweitert werden, dass eine Analyse der einzelnen Messsegmente einer Software-Funktion möglich ist. Das heißt, zusätzlich zu den Messdaten (dynamischen Daten) sollen statische Programmdaten des Kotrollflusses in der Datenbank organisiert werden (s. Abb. 7.8).

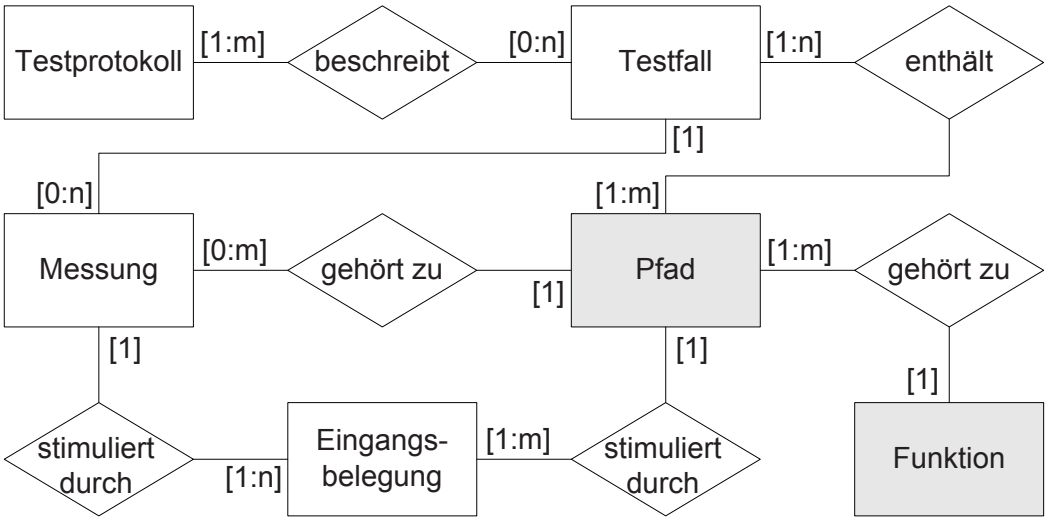


Abb. 7.8: Erweiterung ER-Modell Messdatenorganisation von Funktionen

Messdatenorganisation: Das im Rahmen der programmpfadorientierten Datenanalyse entworfene ER-Modell wird um die Entität ”Funktion” sowie eine ”gehört zu”-Beziehung erweitert, welche zwischen den Entitäten Pfad und Funktion platziert wird. Abbildung 7.8 zeigt das erweiterte ER-Modell der Messdatenorganisation. Da jede Funktion aus mindestens einem Pfad (Teil-Pfad des Prozesses) besteht, ergibt sich die folgende Beziehung der beiden Entitäten:

- 1 bis m Pfade gehören zu einer Funktion.

Wie bereits eingeführt, erfolgt die vollständige Programmpfadbeschreibung erst in der Datenanalyse bei der Datenfusion. Dabei werden die durchlaufenen Programmpfade mit den statischen Pfadinformationen verknüpft. In der Datenorganisation liegen die gemessenen Pfade aus der Pfadverfolgung in der gleichen Tabelle (Entität Pfad) wie die Pfade aus der statischen Codeanalyse. Zur Unterscheidung wird in der Datenablage ein Attribute „Checked“ zur Kennzeichnung der gemessenen Pfade eingeführt (s. Abb. 7.9). Der Funktionsname zeigt auf die statisch ermittelten Programmpfade (die unter einer eigenen Pfad-ID abgelegt werden).

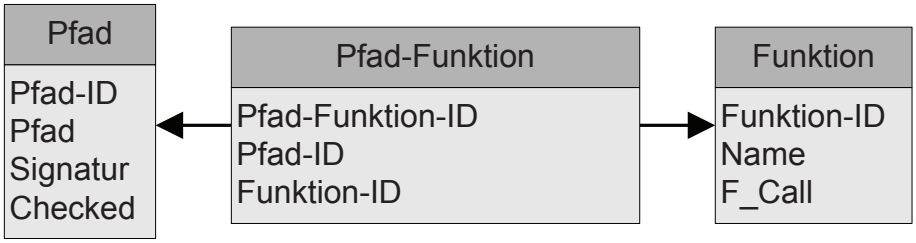


Abb. 7.9: Pfad zu Funktionsbeziehung

- Die Entität Pfad enthält:
  - die Pfadsignatur $pm_{n,i}$ in Form der Sequenz der Zweige $(z_1,...,z_m)$, Messpunkte $(t_1,...,t_m)$ und Funktionsaufrufnamen,
  - den Pfad $p_{n,i}$ als Sequenz der Zweige $(z_1,...,z_m)$ (rekonstruiert aus $pm_{n,i}$),
  - die Kennzeichnung (Checked) für die gemessenen Pfade und
  - die Pfad-ID als eindeutige Nummerierung für jeden unterschiedlichen Programmpfad.
- Die Entität Funktion enthält:
  - die Namen der Funktionen,
  - die Kennzeichnung (F_Call) von aufrufenden und aufgerufenen Funktionen und
  - die Funktion-ID als eindeutige Nummerierung jeder Funktion.

Die Verknüpfung erfolgt über die Beziehung Funktion-Pfad.

- Die Beziehung Funktion-Pfad beschreibt:
  - die Beziehung zwischen Funktion-ID und Pfad-ID.

Struktursuche pfadabhängige BCET: Die im Kapitel 3 vorgestellten Verfahren zur Laufzeitberechnung funktionieren solange, wie die Ausführung des Programmsegmentes nicht unterbrochen wird und die Laufzeiten der Segmentpfade bei jeder Ausführung die gleichen Laufzeiten aufweisen. In Multitask-Umgebungen und bei Architekturen mit Cache und Pipeline sind diese Bedingungen bei der Messung von Segmentlaufzeiten nicht erfüllt. Daher werden hier, wie im Kapitel 6 eingeführt, die Best-Case-Laufzeiten der Segmentpfade zur Bestimmung der theoretischen WCET verwendet. Hierdurch soll der Fehler des Laufzeitmodells begrenzt werden.

7.3.3 Laufzeitmodell

Die Konstruktion der oberen Laufzeitgrenze erfolgt in zwei Schritten, die im Folgenden als Rückwärts- und Vorwärtsmodell[3] beschrieben werden.

Annahme: Nachfolgend wird davon ausgegangen, dass ein Funktionsmodell aus zwei Hierarchieebenen gebildet wird. Der Prozess ist die erste Ebene und die Funktionen (ohne weitere Funktionsaufrufe) die zweite Ebene.

Rückwärtsmodell zur Bestimmung der Hierarchielaufzeit: Das Rückwärtsmodell (s. Abb. 7.10) dient zur Bestimmung der Pfadlaufzeiten einer unmittelbar höheren Funktionsebene.

In dem hier betrachteten Fall von zwei Hierarchieebenen bildet der Prozess die unmittelbar höhere (und höchste) Ebene. Die Berechnung der Pfadlaufzeiten der Hierarchieebene ergibt sich nach Gl. 7.1.

[3] Bezeichnung resultiert aus der Richtung der Berechnungskette

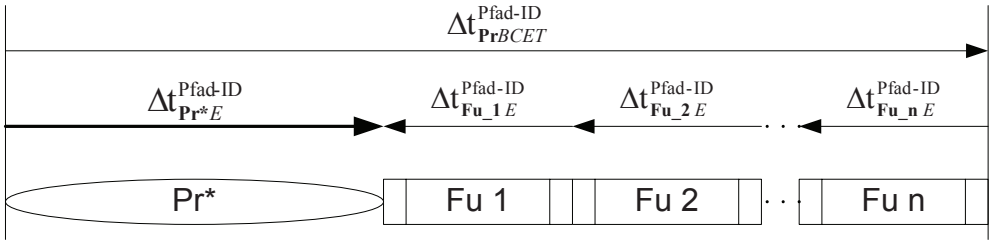


Abb. 7.10: Rückwärtsmodell zur Bestimmung der Hierarchielaufzeit

$$\Delta t^{\text{Pfad-ID}}_{\text{Pr}^*E} = \Delta t^{\text{Pfad-ID}}_{\text{Pr}\,BCET} - \sum_{k=1}^{n} \Delta t^{\text{Pfad-ID}}_{\text{Fu}\_k\,E} \tag{7.1}$$

- $\Delta t^{\text{Pfad-ID}}_{\text{Pr}\,BCET}$ ist die minimale gemessene Laufzeit des Prozesspfades (Pfad-ID, j-te Messung),
- $\Delta t^{\text{Pfad-ID}}_{\text{Fu}\_k\,E}$ ist die Laufzeit der aufgerufenen Funktion k (Messsegmente Fu_*k*) auf dem Pfad des Prozesses (Pfad-ID, j-te Messung) und
- $\Delta t^{\text{Pfad-ID}}_{\text{Pr}^*E}$ ist die Laufzeit des Prozesses (Pfad-ID) ohne Funktionslaufzeiten (Messsegmentlaufzeiten).

Vorwärtsmodell zur Bestimmung der theoretischen WCET: Das Vorwärtsmodell (s. Abb. 7.11) wird zur Bestimmung der theoretischen Worst-Case-Laufzeit einer unmittelbar höheren Funktionsebene benutzt. Dabei werden die maximalen Einzellaufzeiten der Funktionen (auf den Pfaden der unmittelbar höheren Hierarchieebene) betragsmäßig mit den jeweiligen Pfadlaufzeiten der unmittelbar höheren Hierarchieebene addiert.

Das Ergebnis nach Gl. 7.2 liefert die theoretische Worst-Case-Laufzeit für die ungünstigste Kombination von maximalen Einzellaufzeiten, in dem hier betrachteten Fall von zwei Hierarchieebenen.

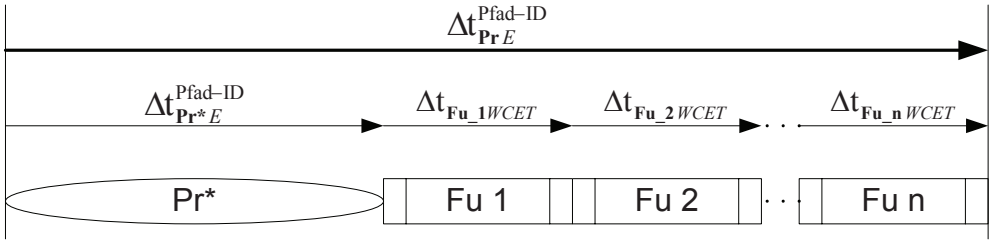


Abb. 7.11: Vorwärtsmodell zur Bestimmung der theoretischen WCET

$$\Delta t_{\text{Pr}\,thWCET} = \max\left(\Delta t^{\text{Pfad–ID}}_{\text{Pr}^*E} + \sum_{k=1}^{n} \Delta t_{\text{Fu}\_kWCET} \right) \tag{7.2}$$

- $\Delta t^{\text{Pfad-ID}}_{\text{Pr}^*E}$ ist die Laufzeit des Prozesspfades (Pfad-ID) ohne Funktionslaufzeiten (Messsegmentlaufzeiten),
- $\Delta t_{\text{Fu}\_k\,WCET}$ ist die maximale Laufzeit der aufgerufenen Funktionen k (Messsegmente Fu_*k*) auf einem Pfad (Pfad-ID) eines Prozesses und
- $\Delta t_{\text{Pr}\,thWCET}$ ist die theoretische Worst-Case-Laufzeit des Prozesses.

7.4 Experimente zur funktionspfadorientierten Datenanalyse

Am Beispiel der Software-Funktion PR_TimMdl (s. Abb. 7.12) soll das Laufzeitmodell erläutert werden. Der Prozess wird hier durch die Funktion PR_TimMdl_10ms() dargestellt. Die Messsegmente sind der Prozess selbst und die Unterfunktion RSFLIPFLOP_RT_U8_-compute(). Die Unterfunktion wird in drei Pfaden des Prozesses aufgerufen. Die Unterfunktionen ht_100(), ht_400() und RSFLIPFLOP_RT_U8_q() mit SFP-Eigenschaften werden nicht einzeln gemessen und gehen mit ihrer Laufzeit in die nächsthöhere Ebene (hier Prozess) ein. In Kapitel 5 wurden bereits die Testfälle mit den Pfad-IDs 2, 5, 7 und 9 aufgestellt. Für die hierarchielokale Pfadabdeckung wird ein weiterer Testfall mit der Pfad-ID 3 gemessen, die Tabelle 7.3 enthält für die fünf Pfade die gemessenen Best-Case-Laufzeiten.

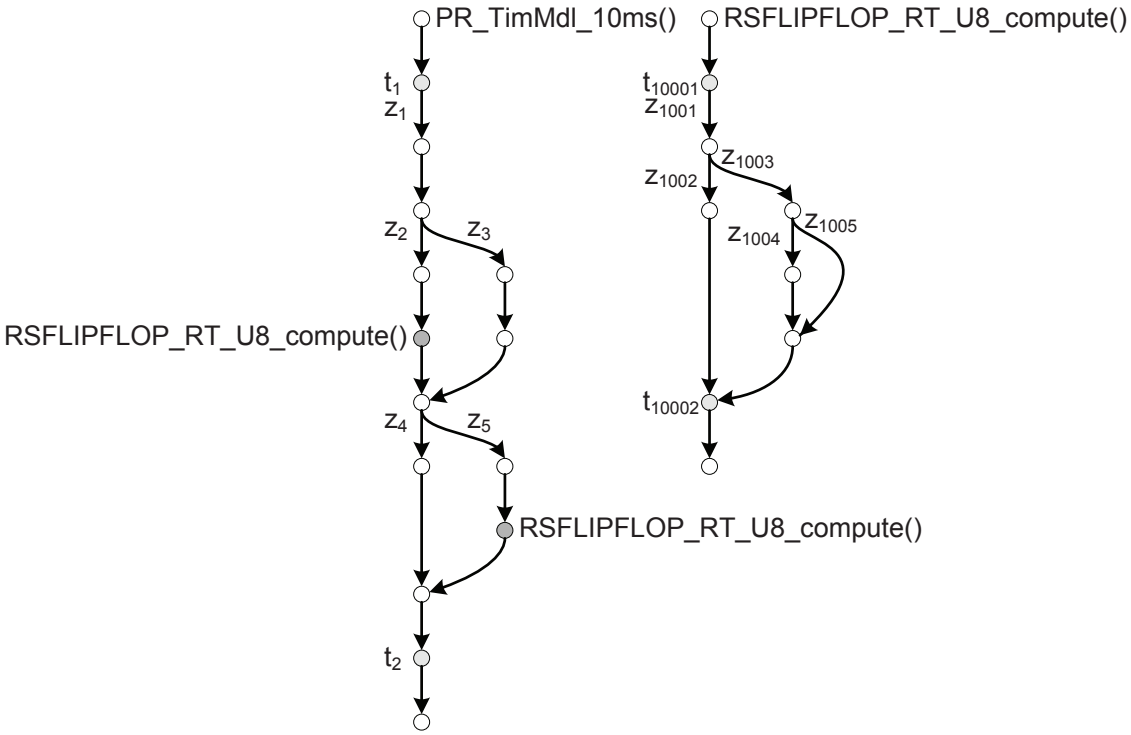


Abb. 7.12: Instrumentierte Funktionen links: PR_TimMdl_10ms(), rechts: Unterfunktion RSFLIPFLOP_RT_U8_compute()

7.4.1 Statische Codeanalyse

Die Tabelle 7.2 zeigt für das Beispielprogramm die tabellarische Aufzählung der Funktionspfadinformationen bezogen auf den Kontrollflussgraphen aus der Abb. 7.12. In der Spalte Pfadsequenz wird eine Folge aus Messpunkten, Zweigen und Funktionsnamen (durch „/“ getrennt) dargestellt, wie sie innerhalb eines möglichen Pfades durchlaufen werden. Um die einzelnen Einträge einer jeden Zeile voneinander unterscheiden zu können, werden die Messpunkte durch ein einleitendes ”t” und die Zweige durch ein einleitendes ”z” kenntlich gemacht. Die Funktionsaufrufe eines Pfades werden durch ein ”fc” (Function Call) gekennzeichnet. Die beschriebenen Merkmale wurden im Software-Visualisierungswerkzeug - CodeGraph - umgesetzt (s. Anhang A.2).

Tabelle 7.2: Aufzählung der Funktionspfadinformationen

Funktionsname	Typ	Pfadsequenz	Pfad
PR_TimMdl_10ms	[F]	/t1/z1/fc_RSFLIPFLOP_RT_U8_compute/z2/z4/t2	/1/2/4
PR_TimMdl_10ms	[F]	/t1/z1/fc_RSFLIPFLOP_RT_U8_compute/z2/fc_RSFLIPFLOP_RT_U8_compute/z5/t2	/1/2/5
PR_TimMdl_10ms	[F]	/t1/z1/z3/z4/t2	/1/3/4
PR_TimMdl_10ms	[F]	/t1/z1/z3/fc_RSFLIPFLOP_RT_U8_compute/z5/t2	/1/3/5
RSFLIPFLOP_RT_U8_compute	[FC]	/t10001/z1001/z1002/t10002	/1001/1002
RSFLIPFLOP_RT_U8_compute	[FC]	/t10001/z1001/z1003/z1004/t10002	/1001/1003/1004
RSFLIPFLOP_RT_U8_compute	[FC]	/t10001/z1001/z1003/z1005/t10002	/1001/1003/1005

Die verschiedenen Einträge der einzelnen Zeilen dieser Tabelle lassen sich wie folgt interpretieren:

- Die erste Spalte zeigt den Namen der untersuchten Funktion.
- In der zweiten Spalte werden diejenigen Funktionen, die in ihrem Pfad noch weitere Funktionen aufrufen, durch ein "[F]" kenntlich gemacht.
- Funktionen, die selbst aufgerufen werden, aber keine weiteren Funktionen mehr aufrufen, werden durch ein "[FC]" (Function Call) kenntlich gemacht.
- In der vorletzten Spalte folgt die Funktionspfadinformation.
- In der letzten Spalte werden die Zweige des jeweiligen Pfades aufgelistet.

7.4.2 Berechnung der theoretischen WCET

Vor der Auswertung der Messreihen (s. Abb. 7.13 und 7.14) erfolgt die Berichtigung der Messergebnisse (s. Kapitel 6). Der bekannte Anteil der systematischen Messabweichung - aus der Korrekturtabelle Messgeräteabweichung (s. Tabelle 6.9) - kann mit umgekehrten Vorzeichen als Korrektur zum Messergebnis $\Delta t_{\mathrm{Pr}\,E}^{Pfad-ID}$ addiert werden. Damit findet man das berichtigte Messergebnis $\Delta t_{\mathrm{Pr}\,KorrE}^{Pfad-ID}$. Dieses weicht vom wahren Wert nur noch um die Summe aus dem unbekannt bleibenden Anteil $e_{s,u}$ der systematischen Messabweichung und der nicht genau feststellbaren zufälligen Messabweichung e_r ab. Zur Berücksichtigung von zufälligen Messabweichungen e_r und der darin enthaltenen zufälligen Messgeräteabweichungen e_{Mr} wird nur die minimale gemessene Laufzeit des Prozessespfades $\Delta t_{\mathrm{Pr}\,BCET}^{Pfad-ID}$ betrachtet (s. Gl. 6.10). Dieser Ansatz vernachlässigt neben zufälligen Abweichungen durch Multitasking-Effekte auch Cache-Fehlzugriffe (s. Abschn. 6.4). Um jedoch auch den ungünstigsten Fall (keine Befehle im Cache) zu betrachten, wird der Worst-Case-Zustand der Hardware hier durch Ausführung des Programmcodes aus dem nicht gecachten Speicher erreicht.

Können Pipelinebelegung, Nebenläufigkeit und Cache-Effekte nicht durch konstruktive Maßnahmen, wie sie in Abschn. 6.3.2 vorgestellt wurden, auf einen vertretbaren und beschreibbaren Anteil reduziert werden, ist eine Ergänzung der Messergebnisse um die Messunsicherheit erforderlich. Beispielsweise kann für die unbekannte systematische Messabweichung durch Pipeline-Effekte die maximale Pipeline-Überlappung zwischen Messobjekt und Messpunkt als Messunsicherheit angegeben werden. Diese einzelnen Messunsicherheiten können als Messabweichungen betrachtet werden. Um die Worst-Case-Abweichung zu bestimmen, werden die maximalen Einzelabweichungen betragsmäßig zu der korrigierten Messobjektlaufzeit (Messsegment) addiert, was zu einer starken Überschätzung der Messobjektlaufzeit führt. Wesentlich hierbei ist jedoch, dass die berechnete obere Laufzeitgrenze sicher ist, d. h., dass die wahre Maximallaufzeit des Programms niemals unterschätzt wird.

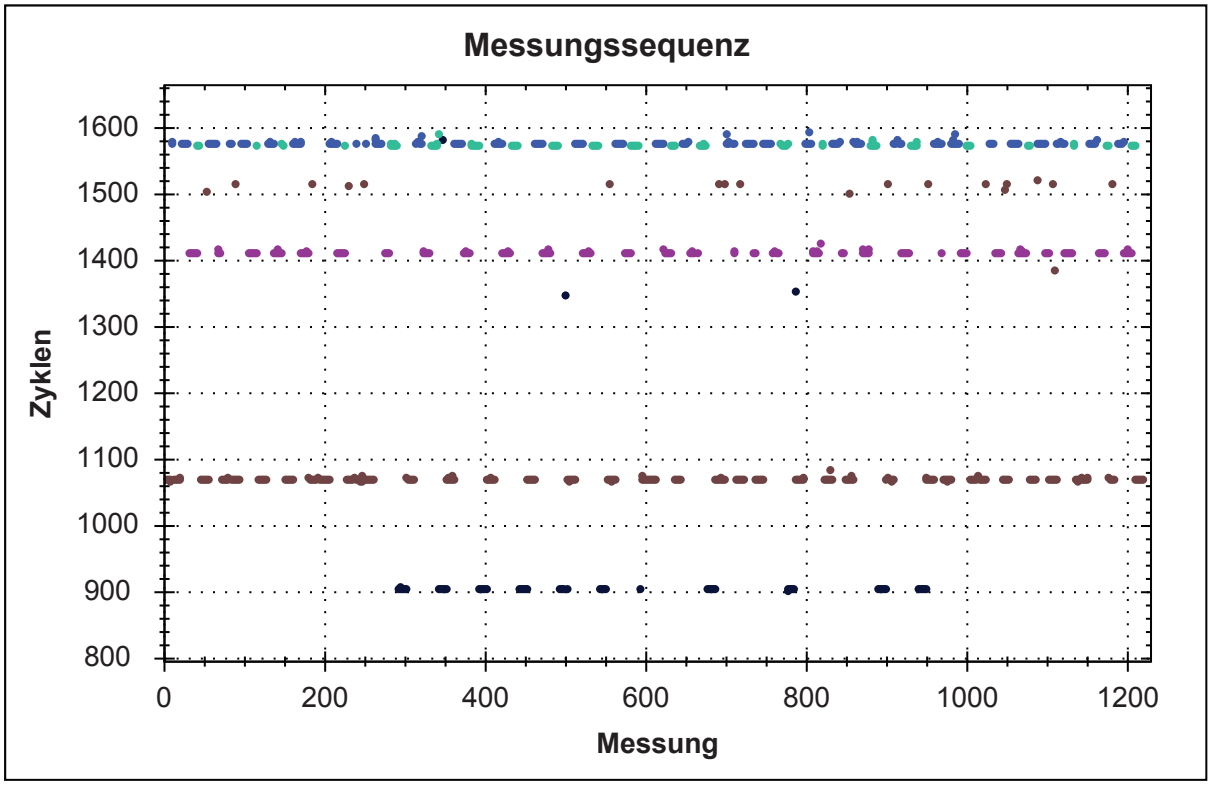


Abb. 7.13: Laufzeitmessung C1 der Funktion PR_TimMdl_10ms() bei n = 4500 min^{-1} ohne Cache

Für die betrachtete Funktion PR_TimMdl_10ms() wurden die Maßnahmen zur Reduktion der unbekannten systematischen Messabweichung nach Abschn. 6.3.2 berücksichtigt. Im Folgenden wird der bekannte Anteil der systematischen Messabweichung durch Korrektion der Messergebnisse berücksichtigt.

Das Laufzeitmodell mit den Gl. 7.1 und 7.2 bleibt gültig, wenn die unberichtigten Messergebnisse $\Delta t_{\mathrm{Pr}\,BCET}^{Pfad-ID}$ und $\Delta t_{Fu\,BCET}^{Pfad-ID}$ durch die berichtigten Messergebnisse $\Delta t_{\mathrm{Pr}\,KorrBCET}^{Pfad-ID}$ und $\Delta t_{Fu\,KorrBCET}^{Pfad-ID}$ ersetzt werden.

Zu den Pfad- und Laufzeitdaten der Funktion PR_TimMdl_10ms() (s. Tabelle 7.3) sind für die Unterfunktion RSFLIPFLOP_RT_U8_compute() auch die Pfad- und Laufzeitdaten (nach Abschn. 7.3) bestimmt worden. In der Abb. 7.14 sind die Laufzeiten der Unterfunktion RSFLIPFLOP_RT_U8_compute() dargestellt.

Tabelle 7.3: Laufzeitmessung C1 der Funktion PR_TimMdl_10ms() bei n = 4500 min^{-1} ohne Cache

Pfad-ID	Pfad	Farbe[a]	gemessene Laufzeit[b]	
			$\Delta t_{\mathrm{Pr}\,BCET}^{Pfad-ID}$	$\Delta t_{\mathrm{Pr}\,KorrBCET}^{Pfad-ID}$
2	/1/2/1001/1003/1005/5/1001/1003/1005		1572	1342
3	/1/3/4		901	855
5	/1/3/5/1001/1003/1004		1409	1271
7	/1/2/1001/1002/4		1066	928
9	/1/2/1001/1003/1004/5/1001/1002		1574	1344

a Darstellung in Abb. 7.13, b Timerzyklen c

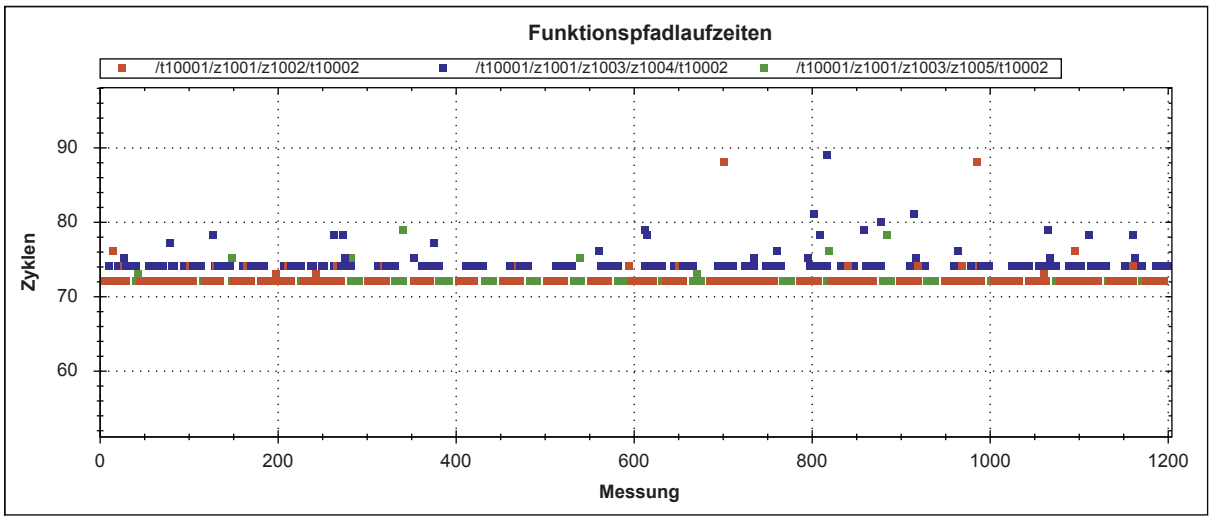


Abb. 7.14: Laufzeitmessung C1 der Unterfunktion RSFLIPFLOP_RT_U8_compute() bei n = 4500 min^{-1} ohne Cache

In der Tabelle 7.4 ist die Laufzeit für den ersten Aufruf der Unterfunktion in den Pfaden mit der ID 2, 5, 7 und 9 angegeben. Der erste Aufruf wird als Position [Fu_1] bezeichnet. In der Tabelle 7.5 ist die Laufzeit für den zweiten Aufruf der Unterfunktion in den Pfaden mit der ID 2 und 9 angegeben, hier mit Position [Fu_2] bezeichnet. Ohne den Einfluss des Caches sind die Laufzeiten der ersten und zweiten Position der Funktionsausführung gleich lang.

Tabelle 7.4: RSFLIPFLOP_RT_U8_compute() [Fu_1]

Pfad-ID[a]	Pfad	gemessene Laufzeit[b] $\Delta t_{Fu\_BCET}^{Pfad-ID}$	$\Delta t_{Fu\_KorrBCET}^{Pfad-ID}$
16	/1001/1002	72	26
17	/1001/1003/1004	74	28
18	/1001/1003/1005	72	26

a Pfad-ID in der Entität Pfad; b Timerzyklen c

Tabelle 7.5: RSFLIPFLOP_RT_U8_compute() [Fu_2]

Pfad-ID[a]	Pfad	gemessene Laufzeit[b] $\Delta t_{Fu\_BCET}^{Pfad-ID}$	$\Delta t_{Fu\_KorrBCET}^{Pfad-ID}$
16	/1001/1002	72	26
17	/1001/1003/1004	n. a.	n. a.
18	/1001/1003/1005	72	26

a Pfad-ID in der Entität Pfad; b Timerzyklen c ; n. a. - nicht ausgeführt

Aus den Pfad- und Laufzeitdaten findet die Rekonstruktion der oberen Laufzeitgrenze für die gesamte Software-Funktion nach Abschn. 7.3.3 statt. Dazu werden die einzelnen Hierarchielaufzeiten der Messsegmente entsprechend dem Rückwärtsmodell berechnet (s. Tabelle 7.6).

Tabelle 7.6: Berechnung der Hierarchielaufzeit

Pfad-ID	Hierarchiepfad	Bezeichnung Pfad-ID/Fu-ID	berechnete Hierarchielaufzeit[a]
2*	1/2/5	2*=2-([Fu_1]+[Fu_2])	1342 – (26 + 26) = 1290
3*	1/3/4	3*=3	855
5*	1/3/5	5*=5-[Fu_1]	1271 – 28 = 1243
7*	1/2/4	7*=7-[Fu_1]	928 – 26 = 902
9*	1/2/5	9*=9-([Fu_1]+[Fu_2])	1344 – (28 + 26) = 1290

a Timerzyklen c

Zu der berechneten Hierarchielaufzeit eines Pfades werden die einzelnen maximalen Funktionspfadlaufzeiten (Messsegmente) nach dem Vorwärtsmodell addiert (s. Tabelle 7.7). Die maximale Laufzeit der aufgerufenen Funktionen k (Messsegmente Fu_*k*) $\Delta t_{\mathrm{Fu\_}k\,WCET}$ wird nach Gl. 6.11 durch Bestimmung der maxBCET ermittelt.

Für die Unterfunktion RSFLIPFLOP_RT_U8_compute() wird die maxBCET für den Pfad mit der ID 17 mit $\Delta t^{Pfad-ID\,17}_{\mathrm{Fu\_1\,max}\,BCET} = \Delta t^{Pfad-ID\,17}_{\mathrm{Fu\_2\,max}\,BCET} = 28\ Zyklen$ ermittelt. Hier wird der Vorteil der funktionspfadorientierten Datenanalyse sichtbar, obwohl für den zweiten Aufruf [Fu_2] dieser Pfad nicht durchlaufen wird, liegt jedoch hierarchielokale Pfadabdeckung durch den ersten Aufruf [Fu_1] vor (s. Tabelle 7.4 und 7.5). Unter den definierten temporalen Bedingungen (kein Cache) kann so auf die Laufzeit für den zweiten Aufruf der Unterfunktion geschlossen werden. Für die Systembibliotheksfunktion RSFLIPFLOP_RT_U8_compute() kann diese Laufzeitinformation auch aus den separat erfassten und in der Laufzeitdatenbank abgelegten Daten abgefragt werden.

Tabelle 7.7: Berechnung der theoretischen Pfadlaufzeiten

Pfad-ID	Prozesspfad	Bezeichnung Pfad-ID/Fu-ID	berechnete Pfadlaufzeit[a]
2*+17+17	/1/2/Fu_1/5/Fu_2	2*+[Fu_1]+[Fu_2]	1290 + 28 + 28 = 1346
3	/1/3/4	3*	855
5	/1/3/5/Fu_1	5*+[Fu_1]	1243 + 28 = 1271
7*+17	/1/2/Fu_1/4	7*+[Fu_1]	902 + 28 = 930
9*+17+17	/1/2/Fu_1/5/Fu_2	9*+[Fu_1]+[Fu_2]	1290 + 28 + 28 = 1346

a Timerzyklen *c*

Für die thWCET wird die maximale Laufzeit aus den berechneten Pfadlaufzeiten (s. Tabelle 7.7) bestimmt und mit der maximalen gemessenen Pfadlaufzeit verglichen (s. Tabelle 7.8).

Tabelle 7.8: Vergleich von berechneter und gemessener maximal Laufzeit

Prozesspfad	Signatur	berechnete thWCET[e]	gemessene maxBCET[e]	temporaler Testfall Eingabe[a]	Ergebnis[b]
/1/2/Fu_1/5/Fu_2	-	1346	-	-	-
/1/2/Fu_1/5/Fu_2	/1/2/1001/1003/1005/5/1001/1003/1005	-	1296[d]	1,0,0,0	0,0 / 1,1[c]

a - Eingabe {B_Fil_0, B_Fil_1, B_Fil_2, B_Fil_3}, b - Ergebnis {B_entpr_1, B_entpr_2}, c - Abhängig vom Grundzustand bzw. Ausgangssituation des Tests, d - korrigierte Laufzeit aus Messung B3 (s. Tabelle 6.13), e Timerzyklen *c*

Das Ziel der hier dargestellten Fallstudie ist es, die Implementierung des temporalen Prüfkonzeptes in Techniken zu demonstrieren, im Anhang A.1 und A.2 werden die verwendeten Werkzeuge vorgestellt. Zur Bewertung der Implementierung soll eine weitere Fallstudie aus dem Anwendungsbereich von Motorsteuergeräte-Software betrachtet werden.

Fallstudie Magnetkupplungssteuerung

Bei der Software-Funktion mit dem zu analysierenden Prozess KUP_10ms() handelt es sich um einen Funktionsbestandteil zur Steuerung der Magnetkupplung für Ottomotoren mit Kompressor. Mittels dieser Magnetkupplung wird bei Bedarf der mechanische Kompressor zugeschaltet. Das Betriebsverhalten der Magnetkupplung lässt sich durch zwei Zustände beschreiben: betätigt und nicht betätigt.

An den Software-Modultest für die Funktion KUP_10ms() werden die gleichen Anforderungen gestellt wie an die zuvor analysierte Funktion PR_TimMdl_10ms(), d. h., es muss eine hierarchielokale Pfadabdeckung erreicht werden. Auf Grundlage der hierfür erstellten Testfälle wird der temporale Test durchgeführt. Die Laufzeitmessung erfolgt auf einem Motorsteuer-

gerät mit TriCore-Prozessor unter definierten Testbedingungen (s. Kapitel 6) mit Funktionsinstrumentierung. Aus den Messergebnissen wird anschließend die obere Laufzeitgrenze für die Funktion berechnet.

Um das Ergebnis der berechneten oberen Laufzeitgrenze zu prüfen, muss die wahre obere Laufzeitgrenze des analysierten Programms bestimmt werden. Hier wird von der richtigen gemessenen oberen Laufzeitgrenze ausgegangen, d. h., von diesem Wert kann angenommen werden, dass er eine vernachlässigbare Abweichung vom wahren Wert der WCET besitzt (s. Kapitel 6). Die gemessene obere Grenze wird durch Bestimmung der maxBCET (s. Gl. 6.11) ermittelt. Hierfür muss eine vollständige Pfadabdeckung durch Testfälle erreicht werden, die Laufzeitmessung hierzu wird im Folgenden als Referenzmessung bezeichnet.

Die Laufzeitbetrachtung in Tabelle 7.9 erfolgt nach Prozessinstrumentierung (s. Abschn. 4.4.1) mit vollständiger Pfadabdeckung als Referenzmessung zur Bestimmung der richtigen gemessenen oberen Laufzeitgrenze und nach Funktionsinstrumentierung (s. Abschn. 7.2.2) mit hierarchielokaler Pfadabdeckung. Die erste Spalte der Prozess- und Funktionsinstrumentierung in Tabelle 7.9 gibt jeweils das gemessene und die zweite Spalte das korrigierte Messergebnis an. Die nächste Spalte enthält die berechnete obere Laufzeitgrenze, die nach dem vorgestellten Laufzeitmodell mit der funktionspfadorientierten Datenanalyse ermittelte wurde. Die letzte Spalte zeigt die Abweichung zwischen der wahren oberen Laufzeitgrenze und der berechneten oberen Laufzeitgrenze.

Die Abweichung wird definiert als: $\eta = (\Delta t_{\mathrm{Pr}\,thWCET} - \Delta t_{\mathrm{Pr}\,Korr\,\max\,BCET}) / \Delta t_{\mathrm{Pr}\,Korr\,\max\,BCET} * 100\%$, wobei $\Delta t_{\mathrm{Pr}\,thWCET}$ die berechnete obere Laufzeitgrenze und $\Delta t_{\mathrm{Pr}\,Korr\,\max\,BCET}$ die gemessene korrigierte obere Laufzeitgrenze aus der Prozessinstrumentierung (Referenzmessung) ist.

Zum Vergleich der Abweichungen wird auch das Beispielprogramm PR_TimMdl_10ms() herangezogen. Die Abweichung wird hier auf Basis der Laufzeitmessergebnisse der durchgeführten temporalen Testfälle aus der Laufzeitmessung B3 bestimmt, d. h., hier liegt keine vollständige Pfadabdeckung als Referenzmessung vor. Anstelle der richtigen gemessenen oberen Laufzeitgrenze wird hier die gemessene maxBCET verwendet. Weitere Fallstudien können den Arbeiten [90, 98, 99, 100] entnommen werden, in denen das in dieser Arbeit entwickelte Verfahren zur Anwendung kam.

Tabelle 7.9: Vergleich der funktionspfadorientierten Datenanalyse mit Laufzeitmessung

Programm	gemessene Laufzeit – Prozessinstrumentierung[c]		gemessene Laufzeit – Funktionsinstrumentierung[c]		berechnete Laufzeit – Laufzeitmodell[c]	
	$\Delta t_{\mathrm{Pr}\,\max\,BCET}^{Pfad-ID}$	$\Delta t_{\mathrm{Pr}\,Korr\,\max\,BCET}^{Pfad-ID}$	$\Delta t_{\mathrm{Pr}\,\max\,BCET}^{Pfad-ID}$	$\Delta t_{\mathrm{Pr}\,Korr\,\max\,BCET}^{Pfad-ID}$	$\Delta t_{\mathrm{Pr}\,thWCET}$	η
KUP_10ms	9200[a]	9154[a]	9603[b]	9373[b]	12312[b]	34,5%
PR_TimMdl	1342[d]	1296	1574	1344	1346	3,9%

a Werte aus [99]; b Werte aus [100]; c Timerzyklen c; d Wert aus Tabelle 6.13

Vergleich zwischen Laufzeitmessung und Laufzeitmodell: Die Ergebnisse in Tabelle 7.9 zeigen, dass die funktionspfadorientierte Datenanalyse mit dem verwendeten Laufzeitmodell sehr unterschiedliche Abweichungen zu den temporalen Testfällen liefern. Wie durch die Referenzmessung aus der Fallstudie Magnetkupplungssteuerung gezeigt wird, kann zwischen der berechneten oberen Laufzeitgrenze und der durch einen Testfall stimulierten und gemessenen oberen Laufzeitgrenze eine hohe Abweichung auftreten (hier über 34%). Im Allgemeinen liegt keine vollständige Pfadabdeckung[4] (wie hier durch die Referenzmessung) vor, so dass die Abweichung auf Basis der Laufzeitmessergebnisse der durchgeführten temporalen

[4] Bei vollständiger Pfadabdeckung kann eine funktionspfadorientierte Datenanalyse entfallen (s. Abschn. 5.3).

Testfälle betrachtet werden muss. Die Funktion PR_TimMdl_10ms() zeigt eine typische Testabdeckung anhand der die Abweichung beurteilt werden muss. Wie in Tabelle 7.9 gezeigt wird, weist der temporale Testfall eine relative kleine Abweichung, hier von 3,9 %, zu der theoretischen oberen Laufzeitgrenze auf. Beide Fälle zeigen ein typisches Problem, dem die statischen Prüftechniken (s. Abschn. 3.2) unterworfen sind, in dem bei der Berechnung der thWCET auch die nichtausführbaren Pfade einbezogen werden. Betrachtet man den in der Tabelle 7.8 angegebenen theoretischen Worst-Case-Pfad der Funktion PR_TimMdl_10ms() so lässt sich dieser aufgrund von Datenabhängigkeiten nicht erreichen, das Gleiche gilt für die Funktion KUP_10ms() in Tabelle 7.9.

Für die Funktion PR_TimMdl_10ms() kann der Tester auf Basis der vorliegenden Daten feststellen, dass die Software-Funktion selbst für die Ausführung aus dem nicht gecachten Speicherbereich die geforderten Rechenzeitrestriktionen von 7500 Zyklen (bzw. 100 µs) im 10 ms Task einhält. Zudem kann zum temporalen Testfall eine Unsicherheit von 3,9 % zur theoretischen oberen Laufzeitgrenze angegeben werden. Bei der Prüfung der Funktion KUP_10ms() auf Einhaltung der in der Spezifikation geforderten Laufzeiteigenschaften kann durch die programmpfadorientierte und funktionspfadorientierte Datenanalyse festgestellt werden, dass die Laufzeitrestriktionen nicht eingehalten werden. Die funktionspfadorientierte Datenanalyse liefert darüber hinaus eine Aussage über die Unsicherheit des temporalen Testfalls.

Abhängig von der Höhe der Abweichung ist der Tester gefordert zu prüfen, ob ein besserer Testfall spezifiziert werden muss. Dies bedeutet jedoch nicht unerheblichen Mehraufwand, der Tester wird dabei unterstützt indem die Funktionen/Unterfunktionen und deren Pfade, die zum Worst-Case beitragen, durch die Werkzeuge ausgegeben werden. Beim Tester bleibt jedoch die Aufgabe, die Ausführbarkeit des Pfades zu bewerten.

7.5 Fazit funktionspfadorientierte Datenanalyse

Die im Kapitel 5 eingeführte programmpfadorientierte Datenanalyse kann vollständige Ergebnisse unter definierten Randbedingungen liefern, wenn die temporalen Testfälle alle Programmpfade abdecken. Um dem Risiko von ungetesteten Programmpfaden zu begegnen, wurde die funktionspfadorientierte Datenanalyse (mit Funktionslaufzeitmessung) eingeführt, hierfür wurden lediglich die minimalen Anforderungen zum Software-Modultest von vollständiger Zweigabdeckung auf hierarchielokale Pfadabdeckung erweitert. In der Praxis hat sich gezeigt, dass die wesentlichen Anforderungen an eine Funktion auf der höchsten Modulebene (im graphischen Funktionsmodell) modelliert werden, und dass hier eine hohe Abdeckung durch die funktionalen Testfälle erreicht wird. Der Aufwand für die Spezifikation von zusätzlichen Testfällen kann hier gering gehalten werden, um eine hierarchielokale Pfadabdeckung zu erreichen. Auf unteren Funktionsebenen dominieren Systembibliotheksfunktionen, deren Laufzeiten vollständig im Vorfeld bestimmt werden. Die Testfälle liefert hier der ASAM MBFS Standard, der zusätzliche Aufwand begrenzt sich lediglich auf die einmalige Messung der Laufzeiten der Systembibliotheksfunktionen unter den jeweiligen Randbedingungen.

Wenn Funktionen/Unterfunktionen sehr viele Pfade enthalten, kommen wieder die Probleme der programmpfadabhängigen Laufzeitmessung zum Tragen, dass die Pfadabdeckung nur mit hohem Aufwand zu realisieren ist. Ein weiterer Nachteil der Laufzeitmessung auf Funktionsebene ergibt sich durch die zusätzlichen Messpunkte im Programmcode, die zulasten der Genauigkeit des Messverfahrens gehen. Jeder Messpunkt bringt neben der bekannten systematischen Messabweichung auch eine unbekannte und zufällige Messabweichung mit sich, die nicht korrigiert werden kann. Hierfür wurden die SFP-Eigenschaften nach [2, 3] so ange-

wendet, dass die Anzahl der Messpunkte über die Hierarchieebenen reduziert werden kann. Eine weitere Fehlerquelle ist die Modellvorstellung auf Funktionsebene. Die Annahme, dass jede Funktion im Prozesspfad mit ihrem WCET in die Gesamtkalkulation eingeht, lässt Datenabhängigkeiten außer Acht und liefert unter Umständen nur weit überschätzte obere Laufzeitgrenzen. Jedoch erlaubt diese Methode eine Beurteilung des gemessenen Worst-Case-Pfades und ermöglicht die Suche nach Funktionsteilen, die einen großen Beitrag zur WCET liefern.

Ein entscheidender Vorteil der funktionspfadorientierten Datenanalyse ist, dass kein Architekturmodell vorhanden sein muss, um eine obere Laufzeitgrenze zu berechnen. Dies unterscheidet diesen Ansatz von den statischen Prüfverfahren, die im Kapitel 3 vorgestellt wurden. Um dies leisten zu können, müssen die Anforderungen, die an den temporalen Test gestellt werden, erfüllt sein. Dazu gehört das Erreichen einer minimalen Testabdeckung (zur Berücksichtigung von Software-Eigenschaften) unter definierten Wiederholbedingungen (zur Berücksichtigung der Hardware-Eigenschaften). Einen weiteren Vorteil bringt dieses Verfahren durch vollständige Automatisierbarkeit [34].

Die Praxistauglichkeit der funktionspfadorientierten Datenanalyse wurde in mehreren Experimenten getestet. Dabei konnte durch Vergleich der richtigen gemessenen oberen Laufzeitgrenze und der berechneten oberen Laufzeitgrenze die Überschätzung festgestellt werden. Es konnte gezeigt werden, dass diese Überschätzung aus sicheren Annahmen resultiert, die getroffen wurden, wie der ungecachten Programmausführung. Eine weitere Verbesserung der Genauigkeit der berechneten oberen Laufzeitgrenze ist zu erreichen, in dem die Ziel- und Einflussgrößen präziser erfasst werden. Das bedeutet, zum einen den Einfluss der Messpunkte aus der Messung herauszuhalten. Zum anderen bedeutet das, eine starke Überschätzung der berechneten oberen Laufzeitgrenze zu verhindern, indem die realen Worst-Case-Situationen der Hardware berücksichtigt werden, wie dem Start der Programmausführung mit leerem Cache (s. Abschn. 6.4). Dafür können ausgewählte temporale Testfälle im Labor, ohne Gefährdung von Mensch oder Technik, als temporaler Regressionstest unter Wiederholbedingungen mit einem Referenzlaufzeitmessgerät erneut ausgeführt werden. Als Anhaltspunkt für eine mögliche Überschätzung kann der Vergleich der maxBCET (s. Tabelle 6.14) aus Messung B2 (mit Cache) und B3 (ohne Cache) herangezogen werden. Die Messungen weisen eine Abweichung von ca. 15% auf, die auf den Einfluss des Caches zurückzuführen sind. Eine weitere Verbesserung ist vor allem durch die Suche nach unmöglichen Programmpfaden zu erreichen, die dann aus der Berechnung herausgehalten werden können. In den betrachten Beispielen bewegt sich diese Abweichung zwischen 3,9% und 34,5% (s. Tabelle 7.9). Eine Programmpfadanalyse ist mitunter sehr aufwendig. In der Praxis muss abgewogen werden, ob der zusätzliche Aufwand zum Auffinden einer knapperen oberen Laufzeitgrenze gerechtfertigt ist.

Die analytische Qualitätssicherung ist durch das hier vorgestellte temporale Prüfkonzept mit programmpfadorientierter und funktionspfadorientierter Datenanalyse in der Lage, Programmfehler zu entdecken. Durch konstruktiven Eingriff kann anschließend die Fehlentwicklung korrigiert werden. Die funktionspfadorientierte Datenanalyse ermöglicht es, den gewählten temporalen Testfall zu bewerten, d. h. zu beurteilen, wie repräsentativ der Testfall mit der gemessenen Laufzeit gegenüber der berechneten theoretischen WCET des Prozesses ist.

8 Zusammenfassung

Im Fokus dieser Arbeit stand die Einführung der Entwicklung von Software-Funktionen durch den Fahrzeughersteller und die damit verbundenen Anforderungen an eine teilweise fremdentwickelte Software für Motorsteuergeräte. Gezeigt wurde im Kapitel 1 die Notwendigkeit, bei der Entwicklung von Software für Echtzeitsysteme neben den funktionalen Eigenschaften auch die nicht funktionalen Eigenschaften, wie die obere Laufzeitgrenze, sicherzustellen. Neue Herausforderungen entstehen bei der verteilten Entwicklung von Software. Hier ist es erforderlich, eine gemeinsame Komponentenbeschreibung zu definieren, damit Ergebnisse der fremdentwickelten Software-Komponenten mit den Ergebnissen der Systemkomponenten zusammengefasst werden können, um so das gesamte System abzusichern. Hierzu zählt die Beschreibung der oberen Laufzeitgrenze bzw. der maximalen Laufzeit der fremdentwickelten Software-Komponenten als Testfall, um eine Nachvollziehbarkeit bei allen Entwicklungspartnern zu ermöglichen.

Das Ziel dieser Arbeit bestand darin, die analytische Qualitätssicherung so zu erweitern, dass durch dynamische, strukturorientierte Tests vollständige und verlässliche Ergebnisse zur Bewertung des temporalen Verhaltens von Fahrzeugfunktionen möglich sind, ohne den Aufwand etablierter Tests wesentlich zu erhöhen. Dafür wurden im Kapitel 2 die Elemente der analytischen Qualitätssicherung analysiert sowie der organisatorische und technische Rahmen, in dem sich die Laufzeitanalyse einbetten muss, betrachtet. Der Stand der Technik zum Software-Test wurde vorgestellt und die Unzulänglichkeiten zusammengefasst. Für die Beurteilung der Gesamtqualität des Softwareproduktes müssen alle Qualitätsmerkmale beim Testen berücksichtigt werden. Hierfür müssen Tests zur Überprüfung der Anforderungen gestellt und deren Eignung beurteilt werden. Dies schließt temporale Testfälle ein.

Im Kapitel 3 wurden bestehende Ansätze bzw. Analyseverfahren zur Bestimmung der oberen Laufzeitgrenze sowie die hierbei auftretenden Probleme beschrieben und diskutiert. Etablierte dynamische (auf Tests basierende) Prüfmethoden zur Laufzeitbestimmung weisen die folgenden Probleme bzw. Unzulänglichkeiten auf:

1. einen sehr hohen Aufwand für die Versuchsplanung,
2. die Schwierigkeit, sowohl Ziel- als auch Einflussgrößen festzulegen und diese in der Untersuchung zu messen, also quantitativ zu erfassen,
3. einen sehr hohen Aufwand für die Versuchsauswertung und
4. die Vollständigkeit der Ergebnisse.

Der eigene Beitrag liegt in der Entwicklung des temporalen Prüfkonzeptes, um die bestehenden Unzulänglichkeiten bei der analytischen Qualitätssicherung des zeitlichen Verhaltens abzustellen. Das Prüfkonzept stellt eine Vorgehensweise zur Lösung der beschriebenen Probleme bereit. Um das zeitliche Verhalten der Software frühzeitig abzusichern, wird die temporale Prüfung im Software-Modultest durchgeführt. Hierzu wird eine Strategie zum temporalen, dynamischen, strukturorientierten Test formuliert und die notwendigen Anforderungen an den Software-Modultest beschrieben. Es wurde als neuer Ansatz ein temporaler Regressionstest eingeführt. Durch die Übernahme der Testfälle aus dem Software-Modultest in einen temporalen Regressionstest entfällt das aufwendige Erstellen von Testfällen. Der hierzu vorgestellte

Ansatz stellt eine Reduktion der Aufwendungen für die Versuchsplanung gegenüber bisherigen auf Tests basierenden Laufzeitmessungen dar.

Ein praxistauglicher Ansatz zur Laufzeitmessung auf dem Zielsystem muss den knappen Ressourcen eines eingebetteten Echtzeitsystems gerecht werden. Hierzu zählen: die Zugänglichkeit von externen Schnittstellen und damit der Übertragungskapazität von Messdaten, die eingeschränkte Manipulationsmöglichkeit der Hardware, um den Worst-Case-Zustand zu provozieren sowie die Begrenzung des Messspeichers zur Zwischenablage von Messdaten. Um das temporale Verhalten von Software des Motorsteuergerätes zu testen, wurden im Kapitel 4 ein Messprozess und eine Testumgebung vorgestellt, mit der die Ziel- und Einflussgrößen objektiv, reproduzierbar und quantitativ auf der Zielarchitektur ermittelt werden können.

Im Kapitel 5 wird als Teil der Prüfstrategie zur Auswertung der Messdaten des temporalen Tests ein eigenständiges Konzept zur Datenanalyse beschrieben. Die Ergebnisse der Prüfstrategie werden durch die Messdatenorganisation strukturiert und einfach dokumentiert. Dabei entsteht eine durchgängige und reproduzierbare Testdokumentation. Das Auffinden der gemessenen pfadabhängigen WCET und BCET wird in der Implementierung durch den Einsatz einer Datenbank ermöglicht, die eine effiziente und strukturierte Analyse der Messdaten zulässt und den Aufwand der Versuchsauswertung reduziert.

Die Methode der programmpfadorientierten Datenanalyse lässt die Überprüfung von Testfällen auf den tatsächlich durchlaufenen Programmpfad und dessen Laufzeit zu. Weiterhin können bei Versuchsfahrten oder am Prüfstand aufgenommene Daten (Fahrprofile) auf ihr Laufzeitverhalten untersucht werden. Der Nachteil des Verfahrens ist, dass für die getestete Software-Funktion nicht sichergestellt ist, dass ein vollständiges Ergebnis vorliegt, wenn nicht alle unterschiedlichen, vollständigen Pfade unter definierten Bedingungen getestet werden. Typischerweise sollte die pfadabhängige Laufzeitbestimmung dann eingesetzt werden, wenn wenige Programmpfade existieren oder wenn Testfälle gezielt bestimmte Programmsituationen überprüfen sollen.

Durch das Messen auf dem Zielsystem werden Pipeline-, Cache- und Multitasking-Effekte mit in die Messung aufgenommen. Eine Bestimmung der hieraus resultierenden Messungenauigkeit erfordert eine genaue Analyse dieser Effekte, was nur durch eine Untersuchung der zum Einsatz kommenden Hardware möglich ist. Parallelität, datenabhängige Befehlsausführungszeit, Pipelining und Speicherhierarchien sind aufwendig zu bestimmen und setzen genaue Kenntnisse der Architektur voraus. Eine Bestimmung der systematischen Messabweichung ist äußerst schwierig und mit sehr hohem Aufwand verbunden. Eine Interpretation der Laufzeitmessergebnisse unter Beachtung der erwähnten Effekte erfolgt im Kapitel 6, wobei zwei Ansätze als Kalibrierungsmöglichkeit angeboten werden. Dafür werden die Zusammenhänge zwischen dem am Messobjekt gemessenen Wert und dem wahren Wert betrachtet. Sind die resultierenden Messabweichungen bekannt, können diese mit in die Auswertung der Messdaten als Korrektion einfließen. Unter der Voraussetzung, dass die gezeigten Maßnahmen zur Reduktion der unbekannten systematischen Messgeräteabweichung erfüllt werden, ist eine Laufzeitmessung mit einer sehr hohen Genauigkeit zu erreichen. Mit dem dargestellten Ansatz wird zudem eine mathematische Analyse von Laufzeitmessreihen möglich, auch wenn die zu messende Software-Funktion durch zufällige Messabweichungen, z. B. durch Multitasking-Effekte, überlagert ist.

Der Nachteil der programmpfadorientierten Laufzeitanalyse ist, dass eine vollständige Pfadabdeckung, wie sie für die sichere Bestimmung des Worst-Case der gesamten Software-Funktion notwendig ist, nur für kleine Programme erreicht werden kann. Bei der Suche nach fehlender Testabdeckung unterstützt das beschriebene Analysesystem, in dem bestimmt wird,

welcher Pfad wie oft durchlaufen bzw. welcher Zweig nicht durchlaufen wurde. Dadurch kann eine gezielte Untersuchung von noch nicht berücksichtigten Programmteilen erfolgen.

Im Kapitel 7 wird eine Ergänzung der programmpfadorientierten Datenanalyse um eine funktionspfadorientierte Datenanalyse vorgeschlagen, um verlässliche und formal vollständige Ergebnisse angeben zu können. Das Ziel der funktionspfadorientierten Datenanalyse ist die Beurteilung von temporalen Testfällen aus der programmpfadorientierten Datenanalyse, um zu bestimmen, wie repräsentativ der gewählte Testfall für die WCET des Prozesses ist. Mit den eingeführten Methoden und Maßnahmen ist dies durch die Angabe einer Unsicherheit zum gewählten Testfall möglich.

Bei dieser Art der Laufzeitanalyse wird der Ansatz verfolgt, die zu messenden Software-Funktionen in Messsegmente aufzuteilen und diese einzeln auf das Laufzeitverhalten zu untersuchen. Als Segmente werden hier Funktionen bzw. Unterfunktionen betrachtet, deren Laufzeitverhalten einmal bestimmt und in weiteren Software-Funktionen wieder verwendet werden kann. Bei der funktionsabhängigen Laufzeitmessung wird nicht mehr versucht eine hohe Pfadabdeckung der gesamten Software-Funktion zu erreichen, vielmehr genügt es, die Pfadabdeckung auf Messsegmentebene zu realisieren. Mithilfe der so gewonnenen Daten kann im Anschluss die maximale Dauer der Ausführungszeiten für die gesamte Software-Funktion rekonstruiert werden. Dazu werden die einzelnen Segmentlaufzeiten addiert.

Um die funktionsabhängige Laufzeitanalyse zu ermöglichen, ist es notwendig, die Kontrollflussstruktur der Messsegmente in die Auswertung der Messdaten einfließen zu lassen. Um dies zu realisieren, wurden im Rahmen dieser Arbeit die Werkzeuge CodeGraph, DataCombiner sowie die zum Ablegen der Messdaten und Pfadinformationen eingesetzte Datenbank entwickelt. Bei CodeGraph handelt es sich um ein Darstellungswerkzeug, das in der Lage ist, den Programmfluss sowie die vom Codegenerator vorgenommenen Instrumentierungen graphisch darzustellen und diese Pfadinformationen für weitere Auswertungen zur Verfügung zu stellen. Die Auswertung der Messdaten und der Pfadinformationen wird durch die Software DataCombiner und der zugrunde liegenden Datenbank erreicht.

Die analytische Qualitätssicherung ist durch das hier vorgestellte temporale Prüfkonzept mit programmpfadorientierter und funktionspfadorientierter Datenanalyse in der Lage, Programmfehler zu entdecken. Durch konstruktiven Eingriff kann anschließend die Fehlentwicklung korrigiert werden.

Literaturverzeichnis

[1] Y. S. Li, S. Malik: Performance Analysis of Real-Time Embedded Software, Kluwer Academic Publishers, 1999

[2] W. Ye: Laufzeitanalyse für Echtzeitprogramme basierend auf Programmpfad-Clustering und Architekturklassifikation, Dissertation, Technische Universität Braunschweig, 2000

[3] F. Wolf: Behavioral Intervals in Embedded Software: Timing and Power Analysis of Embedded Real-Time Software Processes, Kluwer Academic Publishers, 2002

[4] C. L. Liu, J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM 20, S. 46-61, 1973

[5] ETAS GmbH: ERCOSEK V4.2 User´s Guide, Document EC110001 R4.2.7 EN, Stuttgart, 2002

[6] OSEK/VDX: OSEK/VDX Operating System, Version 2.2.3, 2005

[7] Das V-Modell, Entwicklungsstandard für IT-Systems des Bundes, 1997

[8] H. Kuder: Gemeinsames Subset der MISRA C Guidelines, Version 2.0, 2006

[9] U. Brinkschulte, T. Ungerer: Mikrocontroller und Mikroprozessoren, 2. überarbeitete Auflage, Springer Verlag Berlin, 2007

[10] Infineon Technologies AG: TC1796 32-Bit Single-Chip Microcontroller Volume 1 (of 2): System Units, User´s Manual, V0.5, 2005

[11] Infineon Technologies AG: TC1796 32-Bit Single-Chip Microcontroller Volume 2 (of 2): Peripheral Units, User´s Manual, V0.5, 2005

[12] Infineon Technologies AG: TriCore 1 Modular (TC1M) 32-bit Unified Processor, TriCore Pipeline Behaviour & Instruction Execution Timing, Autor: Sorin Zarnescu, Application Note: AP3271a, 2001

[13] A. Spillner, T. Linz: Basiswissen Softwaretest, dpunkt Verlag GmbH, 2004

[14] M. Jersak, K. Richter, R. Ernst: Studie „Software Interfaces“, Institut für Datentechnik und Kommunikationsnetze (IDA), Technische Universität Braunschweig, 2002

[15] M. Jersak, K. Richter, R. Ernst: Studie „Vorschlag für eine zertifizierbare Performanzanalyse eines automotiven Steuergeräts“, Institut für Datentechnik und Kommunikationsnetze (IDA), Technische Universität Braunschweig, 2003

[16] J. Staschulat, J.-C. Braam, M. Jersak, R. Ernst: Studie „Software-Integration für Motorsteuergeräte - Phase 3“, Institut für Datentechnik und Kommunikationsnetze (IDA), Technische Universität Braunschweig, 2004

[17] J. Staschulat, R. Ernst: Studie „Software-Integration für Motorsteuergeräte - Phase 4“, Institut für Datentechnik und Kommunikationsnetze (IDA), Technische Universität Braunschweig, 2005

[18] DIN 44300: Informationsverarbeitung - Begriffe, DIN Deutsches Institut für Normung e.V., Beuth Verlag Berlin, 1988, zurückgezogen

[19] A. Steger: Diskrete Strukturen: Kombinatorik - Graphentheorie - Algebra, Band 1, Springer Verlag Berlin, 1. korrigierter Nachdruck, 2002

[20] R. Storm: Wahrscheinlichkeitsrechnung mathematische Statistik und Qualitätskontrolle, VEB Fachbuchverlag Leipzig, 1976

[21] C.Y. Park: Predicting Program Execution Times by Analyzing Static and Dynamic Program Path, Journal Real Time Systems 5 (1), S. 31-62, 1993

[22] P. Puschner, Ch. Koza: Calculating the Maximum Execution Time of Real-Time Programs, Journal Real Time Systems 1 (2), S. 159-176, 1989

[23] P. Puschner: Zeitanalyse von Echtzeitprogrammen, Dissertation, TU Wien, 1993

[24] P. Scholz: Softwareentwicklung eingebetteter Systeme, Springer Verlag Berlin, 2005

[25] J, Schäuffele, T. Zurawka, Automotive Software Engineering, 3. Auflage, Vieweg Verlag Wiesbaden, 2006

[26] ETAS GmbH: ASCET V5.1 Referenzhandbuch, Dokument EC010005 R5.1.4 DE, Stuttgart, 2005

[27] W. Kalfa: Echtzeit-Betriebssysteme, Fakultät für Informatik, Technische Universität Chemnitz, Skript, SS 2001

[28] P. Liggesmeyer: Qualitätssicherung softwareintensiver technischer Systeme, Spektrum Akademischer Verlag Heidelberg, 2000

[29] DIN 66271: Informationstechnik - Software-Fehler und ihre Beurteilung durch Lieferanten und Kunden, DIN Deutsches Institut für Normung e.V., Beuth Verlag Berlin, 1995

[30] DIN 66272: Informationstechnik - Bewertung von Softwareprodukten, DIN Deutsches Institut für Normung e.V., Beuth Verlag Berlin, 1994, zurückgezogen

[31] J. H. Hennessy, D. A. Patterson: Rechnerarchitektur – Analyse, Entwurf, Implementierung, Bewertung, Vieweg Verlag Braunschweig, 1994

[32] W. Fleisch: Validierung komponentenbasierter Software für Echtzeitsysteme, Aachen, Shaker Verlag, 2003, Dissertation, Universität Stuttgart, 2003

[33] A. Schulze, F. Wolf, W. Daehn: Laufzeitbestimmung von Fahrzeugfunktionen im Motorsteuergerät, 26. Tagung Elektronik im Kraftfahrzeug, Dresden, Expert Verlag, Moderne Elektronik im Kraftfahrzeug, S. 111-125, 2006

[34] A. Schulze, J. P. Achnitz: Effizienzsteigerung bei der Verifikation nichtfunktionaler zeitkritischer Anforderungen von Fahrzeug-Funktionen der Motorsteuerung unter Nutzung der HiL-Simulation, Tagungsband 8. Tagung Hardware-in-the-Loop-Simulation, 2008

[35] AUTOSAR GbR: AUTOSAR Technical Overview V2.2.1 R3.0, 2008

[36] DIN 1319-1: Grundlagen der Messtechnik - Teil1: Grundbegriffe, DIN Deutsches Institut für Normung e.V., Beuth Verlag Berlin, 1995

[37] R. Moser, T. Reck: ASAM AE Model Based Function Specification – Standard Blockset, Publikation, Version 1.0, Association for Standardisation of Automation and Measuring Systems (ASAM e.V.), Höhenkirchen, 2006

[38] F. Wolf, A. Schulze: A Distributed Design and Test Methodology for Automotiv Systems, Proceedings embedded world Conference 2007, Nürnberg, 2007

[39] D. Zöbel: Echtzeitsysteme - Grundlagen der Planung, Springer Verlag Berlin, 2008

[40] W. A. Halang, R. Konakovsky: Sicherheitsgerichtete Echtzeitsysteme, Oldenbourg Industrieverlag GmbH München, 1999

[41] R. Moser: Modellbasierte Software-Entwicklung für verteilte Echtzeitsysteme im Kraftfahrzeug, Dissertation, Universität Stuttgart, Expert Verlag, 2006

[42] ASAM MCD-2: Association for Standardisation of Automation- and Measuring Systems (ASAM), ECU Measurement and Calibration Data Exchange Format, V 1.6.1

[43] D. Huhnke: Skript GEM - Grundlagen der elektrischen Messtechnik, Institut für Elektrische Messtechnik, Technische Universität Braunschweig, (Stand: 27.10.2007)

[44] K. Reif: Automobilelektronik - Eine Einführung für Ingenieure, Vieweg Verlag Wiesbaden, GWV Fachverlage GmbH, 2007

[45] J. Fenlaso, R. Stallman: GNU gprof - The GNU Profiler, 1997

[46] R. Stallman: Using the GNU Compiler Collection (GCC), Last updated 23 May 2004 for GCC 3.4.6, 2004

[47] HighTec EDV-Systeme GmbH: User’s Guide HighTec GNU Toolchain for TriCore, Version 1.14, Saarbrücken, 2006

[48] J. Staschulat, R. Ernst, A. Schulze, F. Wolf: Context Sensitive Performance Analysis of Automotive Applications, Proceedings of the conference on Design, Automation and Test in Europe - Volume 3, S. 165-170, 2005

[49] R. Dumke, E. Foltin, R. Koeppe, A. Winkler: Softwarequalität durch Meßtools - Assessment, Messung und instrumentierte ISO 9000, Vieweg Verlag Wiesbaden, 1996

[50] W. Ye, R. Ernst, T. Benner, J. Henkel: Fast Timing Analysis for Hardware-Software Co-Synthesis, Proc. of ICCD, IEEE Society Press, S. 452-457, 1993

[51] M. Heide: Performance Tuning, Profiling und Werkzeuge in Java, Einführung, 2005, URL: http://www.plm.eecs.uni-kassel.de/plm/fileadmin/pm/courses /optiProg05/ progOptiAusarb1.pdf, Stand: 14.12.2009

[52] O. Busse: Verfahren und Tools zur Bestimmung der Worst-Case Laufzeit (WCET) von Softwarekomponenten im Motorsteuergerät, Recherche, Beschreibung und Bewertung für die Praxis, Studienarbeit, Fachhochschule Braunschweig/Wolfenbüttel, Fachbereich Maschinenbau, 2008

[53] S. M. Petters: Comparison of Trace Generation Methods for Measurement Based WCET Analysis, In Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis, 2003

[54] P. Liggesmeyer: Software-Qualität - Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag Heidelberg, 2002

[55] S. M. Petters: Bounding the Execution Time of Real-Time Tasks on Modern Processors, In Proceedings of the 7th International Conference on Real–Time Computing Systems and Applications, 2000

[56] S. M. Petters: Worst Case Execution Time Estimation for Advanced Processor Architectures, Dissertation, Technischen Universität München, Fakultät für Elektrotechnik und Informationstechnik, 2002

[57] S. M. Petters, G. Färber: Making Worst Case Execution Time Analysis for Hard Real–Time Tasks on State of the Art Processors Feasible, In Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications, 1999

[58] Infineon Technologies AG: An Easy Way to Measure TriCore Cycle Performance, Autor: Lance Zheng, Application Note: AP32075, 2002

[59] iSYSTEMS: Execution Coverage - User´s Guide V9.7.155, 2008

[60] iSYSTEMS: Profiler - User´s Guide V9.7.155, 2008

[61] ETAS GmbH: RTA-TRACE Getting Started Guide, Dokument TD00001-004, Stuttgart, 2004

[62] R. Wilhelm, J. Engblom u. a.: The Worst-Case Execution Time Problem– Overview of Methods and Survey of Tools, ACM Transactions on Embedded Computing Systems (TECS), S. 1-53, 2008

[63] P. Puschner, R. Nossal: Testing the Results of Static Worst-Case Execution-Time Analysis, In Proceedings of the 19th IEEE Real-Time Systems Symposium, S. 134-143, 1998

[64] iSYSTEMS: Trace Getting Started, Trace using an In-Circuit Emulator, V8.02, 2003

[65] R. Kirner, P. Puschner, I. Wenzel: Measurement-Besed Worst-Case Execution Time Analysis using Automatic Test-Data Gerneration, In Proc. IEEE Workshop on Software Tech. for Future Embedded and Ubiquitous Systs. (SEUS'05), S. 7-10, 2004

[66] A. von Bülow: Optimale Cache-Nutzung für Realzeitsoftware auf Multiprozessorsystemen, Dissertation, Technische Universität München, Fakultät für Elektrotechnik und Informationstechnik, 2005

[67] A. Ermedahl, B. Lisper u. a.: Static WCET analysis - General overview-, Präsentation, MRTC der Mälardalen University, Schweden, 2007, URL: http://www.mrtc.mdh.se/events/files/ID1296.pdf, Stand: 14.12.2009

[68] J. Gustafsson, B. Lisper u. a.: A Tool for Automatic Flow Analysis of C-programs for WCET Calculation, In 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), 2003

[69] H. Theiling, C. Ferdinand: Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis, In Proceedings of the IEEE Real-Time Systems Symposium (RTSS'98), S. 144-153, 1998

[70] T. Lundqvist: A WCET Analysis Method for Pipelined Microprocessors with Cache Memories, Doktorarbeit, Chalmers University of Technology, Göteborg (Schweden), Department of Computer Engineering, 2002

[71] Mälardalen Real-Time Research Center (MRTC): WCET project / SWEET, URL: http://www.mrtc.mdh.se/projects/wcet/sweet.html, Stand: 14.12.2009

[72] R. Heckmann, C. Ferdinand: Worst-Case Execution Time Prediction by Static Program Analysis, In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), S. 26-30, 2004

[73] Rapita Systems Ltd: pWCET White Paper, Edition 1, DOC-040205-2, York, 2004

[74] Tudorum Ltd.: Bound-T Assertion Language, Version 6.4, 2010

[75] Tudorum Ltd.: Bound-T timing analysis tool - User Guide, Version 6.3, 2009

[76] L. Tan: The Worst Case Execution Time Tool Challenge 2006, Technical Report for the External Test, In Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06), 2006

[77] H. Wallentowitz, K. Reif: Handbuch Kraftfahrzeugelektronik, Wiesbaden, Vieweg Verlag, GWV Fachverlage GmbH, 2006

[78] A. Schulze: Laufzeitanalyse für höhere Software-Funktionen im Motorsteuergerät am Beispiel Geschwindigkeitsregelanlage (GRA), Diplomarbeit, Hochschule Magdeburg-Stendal (FH), 2003

[79] C. H. Cap: Theoretische Grundlagen der Informatik, Springer Verlag Wien, 1993

[80] F. Stetter: Grundbegriffe der Theoretischen Informatik, Springer Verlag Berlin, 1988

[81] DIN 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/ elektronischer/ programmierbarer elektronischer Systeme, Teile 1 bis 7, DIN Deutsches Institut für Normung e.V., Beuth Verlag Berlin, 2002

[82] A. Zeller: Software-Metriken, Lehrstuhl Softwaretechnik, Universität des Saarlandes, Saarbrücken, 2002

[83] D. Lichtenthäler, J. P. Achnitz, M. Brand, F.Wolf: Parametrierung von echtzeitfähigen Simulationsmodellen für Motoren mit Abgas-Turbolader, Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik 2, Expert Verlag, S. 42-65, 2008

[84] K. Frühauf, J. Ludewig, H. Sandmayr: Software-Prüfung - Eine Anleitung zum Test und zur Inspektion, vdf Hochschulverlag AG, 2007

[85] E. H. Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme, Teubner Verlag, 1997

[86] B. Beizer: Software Testing Techniques, International Thomson Computer Press, 2. Auflage, 1990

[87] A. Wohnhaas, R. Moser: Modellaustausch zwischen Steuergeräte-Entwicklungstools auf Basis einheitlicher grafischer Blockbibliotheken, 3. Stuttgarter Symposium Kraftfahrwesen und Verbrennungsmotoren, Expert Verlag, 1999

[88] H. Balzert: Lehrbuch der Software-Technik, 2. Auflage, Spektrum Akademischer Verlag, 2000

[89] S. Diehl: Softwarevisualisierung, Informatik-Spektrum, Volume 26, Nr. 4, S. 257-260, Springer Berlin/Heidelberg, 2003

[90] O. Busse: Entwicklung und Evaluierung einer Laufzeitmessumgebung für Bibliotheksfunktionen des Motorsteuergerätes, Diplomarbeit, Fachhochschule Braunschweig/ Wolfenbüttel, Fachbereich Maschinenbau, 2008

[91] ISO 11898: Road vehicles - Controller area network (CAN), International Organization for Standardization (ISO), 2003

[92] ISO 9141: Road vehicles – Diagnostic systems – Requirements for interchange of digital information, International Organization for Standardization (ISO), 1989

[93] IEEE-ISTO 5001-2003: The Nexus 5001 Forum - Standard for a Global Embedded Processor - Debug Interface, Institute of Electrical and Electronics Engineers (IEEE), 2003

[94] IEEE Std. 1149.1-1990: Joint Test Action Group (JTAG), Standard Test Access Port and Boundary Scan, Institute of Electrical and Electronics Engineers (IEEE), 1990

[95] ETAS GmbH: ETKS4.1 Emulator Probe for Infineon TC1766 and TC1796 Data Sheet, Dokument QH110524 R1.0.2 EN, Stuttgart, 2006

[96] ETAS GmbH: ETKT2.0 Emulator Probe for Infineon TC1792 and TC1796 / TC1796ED Data Sheet, Dokument QH110529 R1.0.2 EN, Stuttgart, 2006

[97] A. Heuer, G. Saake: Datenbanken: Konzepte und Sprachen, 2. aktualisierte und erweiterte Auflage, International Thomson Publishing, Bonn, 2000

[98] J. Hennig: Durchführung einer Laufzeitanalyse und Darstellung der Pfadabdeckung anhand einer Beispielfunktion der Motorsteuergerätesoftware, Diplomarbeit, Fachhochschule Braunschweig/Wolfenbüttel, Fachbereich Fahrzeug-, Produktions- und Verfahrenstechnik, 2008

[99] M. Dübner: Analyse und Darstellung pfadabhängiger Laufzeit-Messdaten von Fahrzeugfunktionen der Motorsteuergerätesoftware, Diplomarbeit, Hochschule für Technik und Wirtschaft Dresden (FH), Fachbereich Informatik/Mathematik, 2006

[100] M. Neubauer: Durchführung einer Performanzanalyse zur Bestimmung von Laufzeiten innerhalb eines OSEK-Echtzeitbetriebssystems auf einem Motorsteuergerät, Diplomarbeit, Universität Lüneburg, Fakultät III Umwelt und Technik, 2007

[101] A. Dietrich: Statische Softwarevisualisierung zur Darstellung der Testtiefe von Motorsteuergeräte-Funktionen, Diplomarbeit, Hochschule Anhalt, 2006

A Anhang

A.1 Aufbau des Datenanalyse- und Dokumentationswerkzeuges

Die im Rahmen der vorliegenden Arbeit beschriebene Versuchsauswertung zur temporalen Prüfung von Software-Funktionen wurde in ein Datenanalyse- und Dokumentationswerkzeug - DataCombiner - umgesetzt. Das Werkzeug ist in der Lage, die unterschiedlichen Datenquellen aus Laufzeitmessung, Pfadverfolgung und statischer Codeanalyse zu kombinieren und zu analysieren. Die Abbildung A.1 zeigt die linke Seite des Analyse-Dialoges, den Auswahlteil.

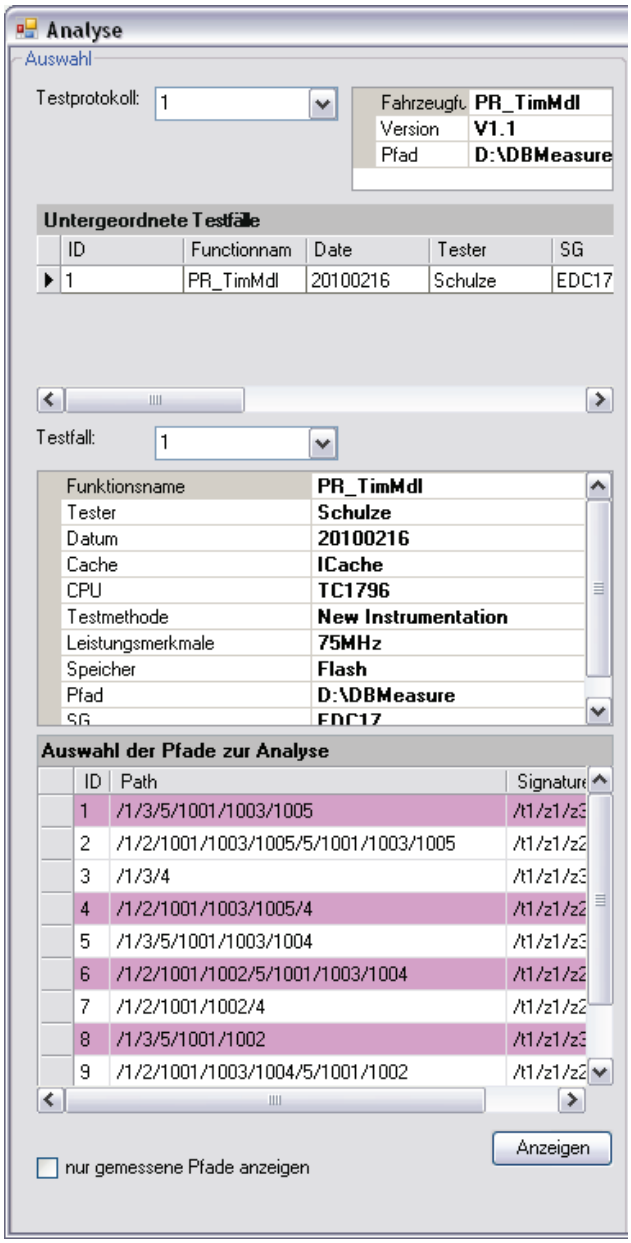


Abb. A.1: Analyse-Dialog DataCombiner - Auswahlteil

Im Auswahlteil kann das Testprotokoll und der dazugehörige Testfall ausgewählt werden. Zu dem gewählten Testfall werden die zugehörigen Pfade dargestellt, hier müssen die zu analysierenden Pfade selektiert werden.

Die Laufzeiten der gemessenen Pfade können in den Diagrammen (Messungssequenz, WCET/BCET Pfad-Übersicht, Histogramm usw.) analysiert werden. Die Abbildung A.2 zeigt die rechte Seite des Analyse-Dialoges (Darstellungsteil). In der oberen Hälfte der Abbildung befinden sich die Reiter zur Auswahl der jeweiligen Diagrammart. In der unteren Hälfte der Abbildung befindet sich die Kurzübersicht über den ausgewählten Pfad.

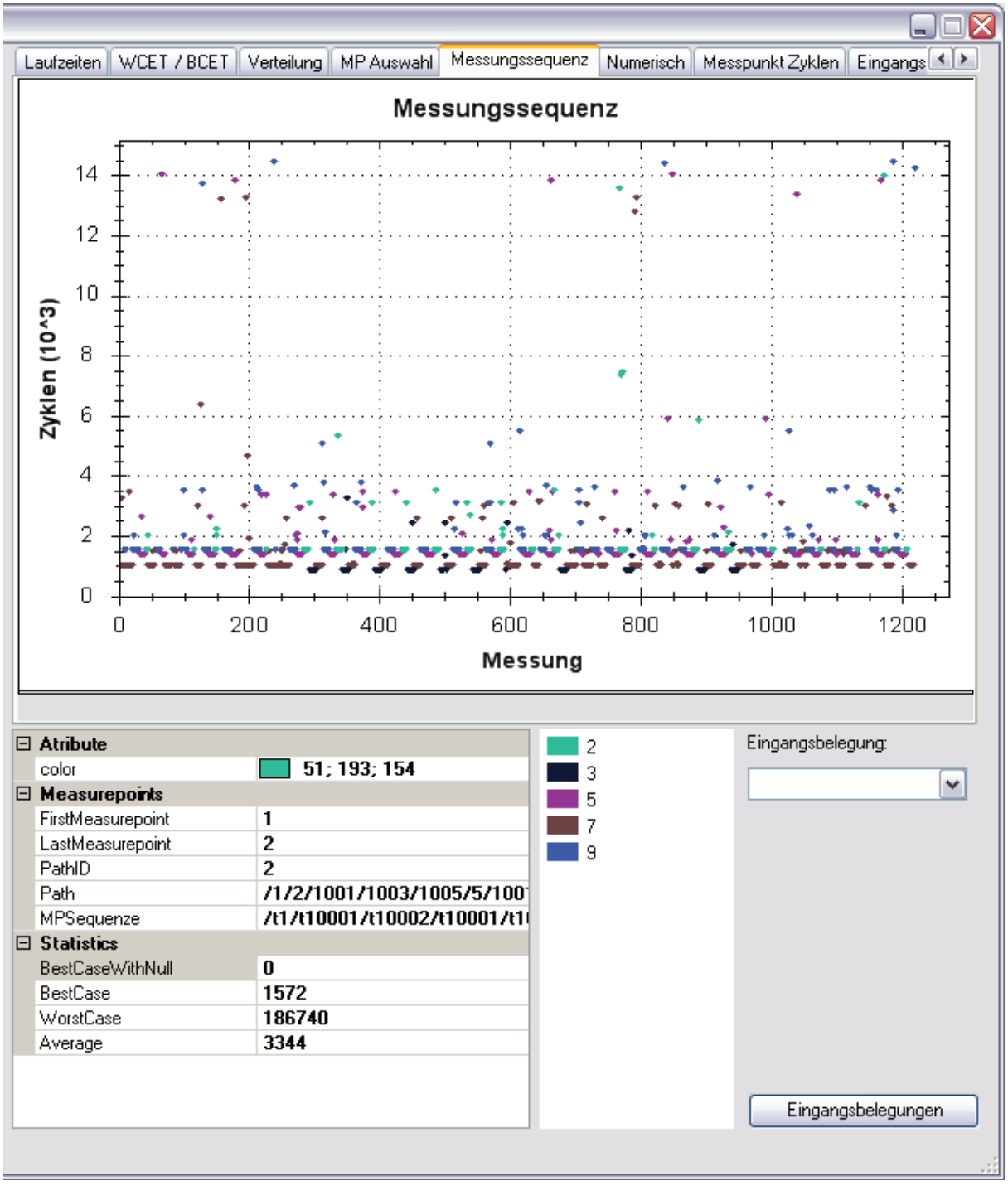
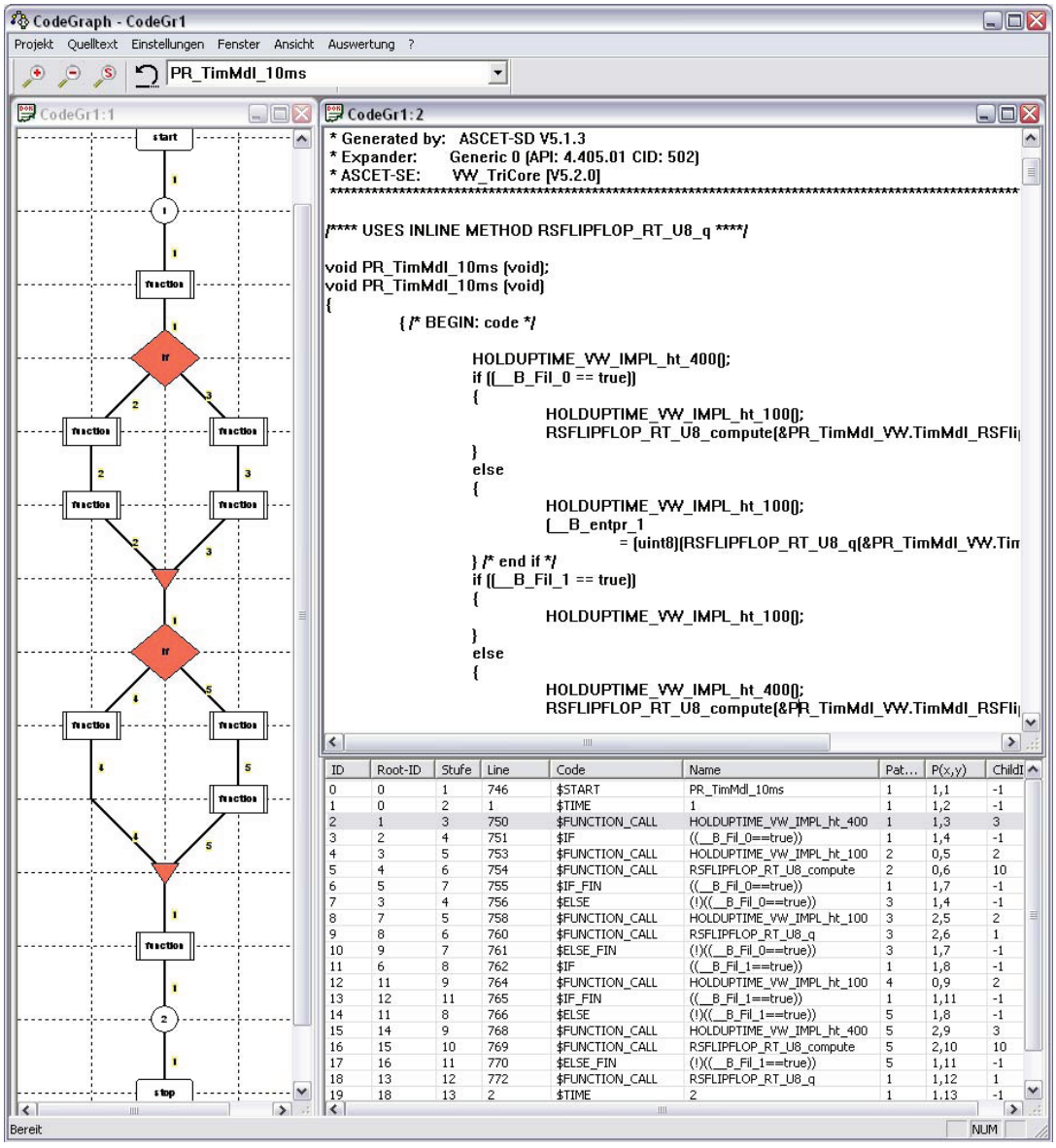


Abb. A.2: Analyse-Dialog DataCombiner – Darstellungsteil

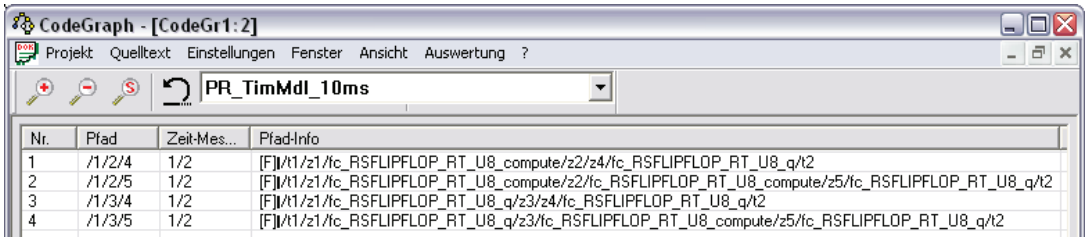
A.2 Aufbau des Software-Visualisierungswerkzeuges

Die im Abschnitt 7.3.1 beschriebenen Merkmale wurden im Software-Visualisierungswerkzeug - CodeGraph - umgesetzt [101], das in der Lage ist, den Kontrollflussgraphen sowie die vorgenommenen Instrumentierungen des C-Codes einer Funktion graphisch darzustellen. Abbildung A.3 zeigt die Hauptansicht mit dem Kontrollfluss einer instrumentierten Funktion. Zur Ausgabe der Funktionspfadinformationen steht der Pfadübersichtsdialog zur Verfügung, in dem alle möglichen Pfade der jeweiligen Funktion ausgegeben werden (s. Abb. A.4).



ID	Root-ID	Stufe	Line	Code	Name	Pat...	P(x,y)	ChildI
0	0	1	746	$START	PR_TimMdl_10ms	1	1,1	-1
1	0	2	1	$TIME	1	1	1,2	-1
2	1	3	750	$FUNCTION_CALL	HOLDUPTIME_VW_IMPL_ht_400	1	1,3	3
3	2	4	751	$IF	((__B_Fil_0==true))	1	1,4	-1
4	3	5	753	$FUNCTION_CALL	HOLDUPTIME_VW_IMPL_ht_100	2	0,5	2
5	4	6	754	$FUNCTION_CALL	RSFLIPFLOP_RT_U8_compute	2	0,6	10
6	5	7	755	$IF_FIN	((__B_Fil_0==true))	1	1,7	-1
7	3	4	756	$ELSE	(!)((__B_Fil_0==true))	3	1,4	-1
8	7	5	758	$FUNCTION_CALL	HOLDUPTIME_VW_IMPL_ht_100	3	2,5	2
9	8	6	760	$FUNCTION_CALL	RSFLIPFLOP_RT_U8_q	3	2,6	1
10	9	7	761	$ELSE_FIN	(!)((__B_Fil_0==true))	3	1,7	-1
11	6	8	762	$IF	((__B_Fil_1==true))	1	1,8	-1
12	11	9	764	$FUNCTION_CALL	HOLDUPTIME_VW_IMPL_ht_100	4	0,9	2
13	12	11	765	$IF_FIN	((__B_Fil_1==true))	1	1,11	-1
14	11	8	766	$ELSE	(!)((__B_Fil_1==true))	5	1,8	-1
15	14	9	768	$FUNCTION_CALL	HOLDUPTIME_VW_IMPL_ht_400	5	2,9	3
16	15	10	769	$FUNCTION_CALL	RSFLIPFLOP_RT_U8_compute	5	2,10	10
17	16	11	770	$ELSE_FIN	(!)((__B_Fil_1==true))	5	1,11	-1
18	13	12	772	$FUNCTION_CALL	RSFLIPFLOP_RT_U8_q	1	1,12	1
19	18	13	2	$TIME	2	1	1,13	-1

Abb. A.3: Software-Visualisierungswerkzeug - CodeGraph



Nr.	Pfad	Zeit-Mes...	Pfad-Info
1	/1/2/4	1/2	[F]/t1/z1/fc_RSFLIPFLOP_RT_U8_compute/z2/z4/fc_RSFLIPFLOP_RT_U8_q/t2
2	/1/2/5	1/2	[F]/t1/z1/fc_RSFLIPFLOP_RT_U8_compute/z2/fc_RSFLIPFLOP_RT_U8_compute/z5/fc_RSFLIPFLOP_RT_U8_q/t2
3	/1/3/4	1/2	[F]/t1/z1/fc_RSFLIPFLOP_RT_U8_q/z3/z4/fc_RSFLIPFLOP_RT_U8_q/t2
4	/1/3/5	1/2	[F]/t1/z1/fc_RSFLIPFLOP_RT_U8_q/z3/fc_RSFLIPFLOP_RT_U8_compute/z5/fc_RSFLIPFLOP_RT_U8_q/t2

Abb. A.4: Pfadübersicht - CodeGraph

A.3 Fallstudie Funktion PR_TimMdl

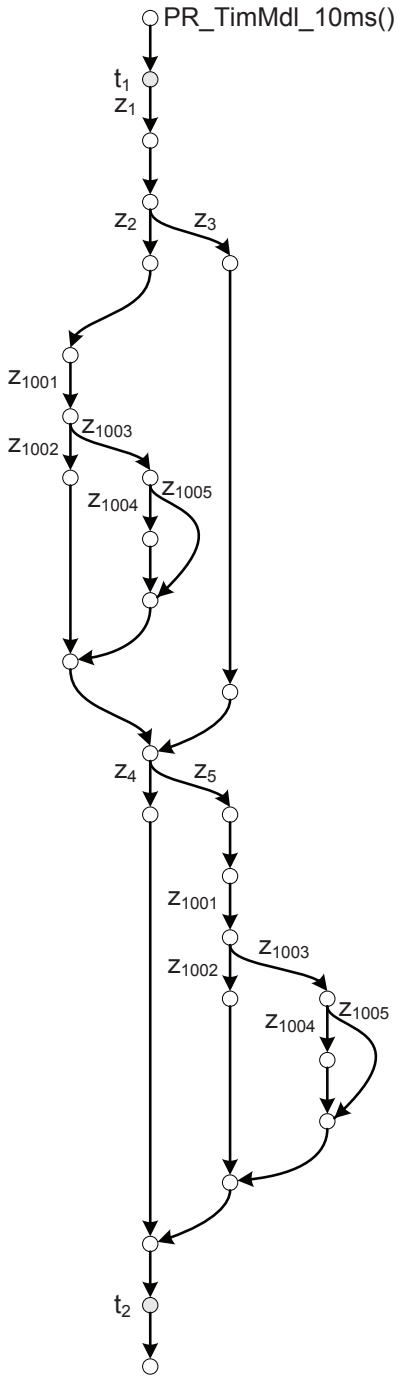


Abb. A.5: Kontrollflussgraph der Funktion PR_TimMdl_10ms()

Tabelle A.1: Metriken der Funktion PR_TimMdl_10ms()

Nr.	Name	AVGS	STMT	VG	GOTO	PARA	PATH	RETU	NBC	LEVL	DC_C
1	PR_TimMdl_10ms	4,55	11	3	0	0	4	0	1	2	4
2	RSFLIPFLOP_RT_U8_compute	4,25	4	3	0	3	3	0	3	3	0
3	ht_100	**	0	1	0	0	1	0	1	1	0
4	ht_400	**	0	1	0	0	1	0	1	1	0
5	RSFLIPFLOP_RT_U8_q	*	*	*	*	*	*	*	*	*	*

* Inline; ** Assembler; NBC NBCALLING; DC_C DC_CALLING

A.4 ASAM Standard Blockset Beschreibung RSFlipFlop

Die formale Beschreibung eines Basisblocks in der ASAM MBFS-Notation [37] lautet:

Verbal description

The second input *u2* (reset) dominates the first input *u1* (set). The first output returns the stored boolean input value, whilst the second output value is the logical negation of the first output.

$$y1(0) = false \tag{A.1}$$

$$y1(n) = \begin{cases} false & : u2(n) == true \\ true & : u2(n) == false \;\&\&\; u1(n) == true \end{cases} \tag{A.2}$$

$$y2(0) = !\,y1(n) \tag{A.3}$$

Icon and variables

Icon	Variables			
	Inputs	Outputs	States	Temporary
	1 u1: logic 2 u2: logic	1 y1: logic 2 y2: logic	1 x: logic	

Pseudocode

System init code	Runcode

x = 0;	if (u2) {
	x = 0;
	}
	else if (u1) {
	x = 1;
	}
	y1 = x;
	y2 = !x;

Test cases

Test data

Time vector:

t 0 1 2 3 4 5 6 7

Input values:

u1 0 0 0 0 1 1 1 1

u2 1 1 0 0 0 0 1 1

Output/state values:

x 0 0 0 0 1 1 0 0

y1 0 0 0 0 1 1 0 0

y2 1 1 1 1 0 0 1 1